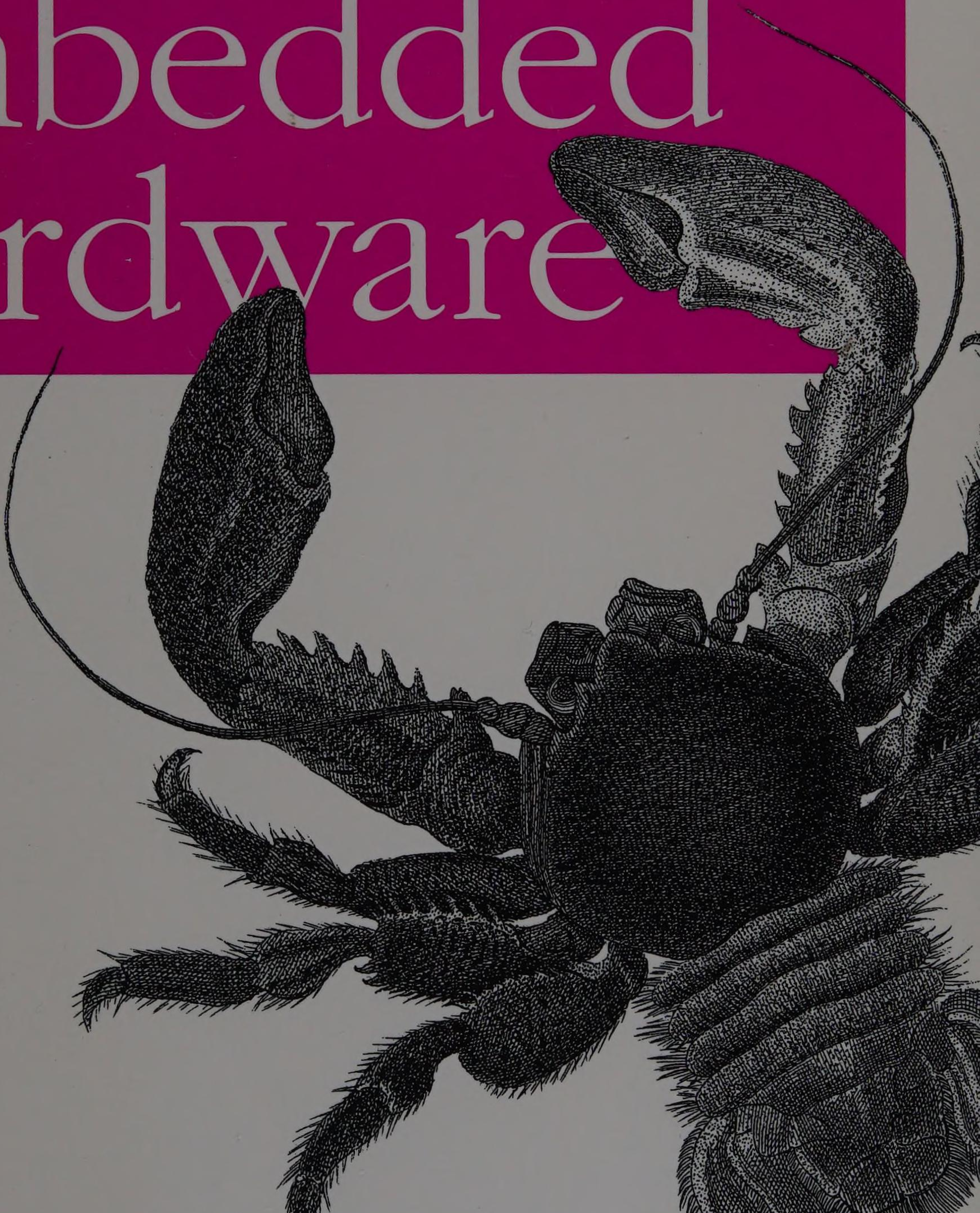
Building Your Own Specialized Computer

Designing
Embedded Hardware



O'REILLY®

John Catsoulis

Designing Embedded Hardware

Designing Embedded Hardware

John Catsoulis

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Designing Embedded Hardware
by John Catsoulis

Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor:	Jon Orwant
Production Editor:	Philip Dangler
Cover Designer:	Emma Colby
Interior Designer:	David Futato
Production Services:	Argosy

Printing History:

November 2002: First Edition.

ISBN: 0-596-00362-5
[M]

This book is dedicated to Kristine, Duncan, Andrew, and James.

Table of Contents

Foreword xi

Preface xiii

Part I. Background

1. Introduction to Computer Architecture 3
Concepts 4
Memory 22
Input/Output 26
DMA 26
Embedded Computer Architecture 32

2. Electronics 101 36
Voltage and Current 36
Analog Signals 38
Power 39
Resistors 39
Capacitors 43
RC Circuits 45
Inductors 49
Transformers 51
Diodes 52
Crystals 55
Digital Signals 59
Understanding Schematics 61
Read the Datasheet 66

3. **Power Sources** **68**
Juice from the Wall 68
Batteries 69
Regulators 70

4. **Building It** **77**
Avoid Noise 77
Quick-and-Dirty Construction 82
Printed-Circuit Boards 83
Routing a Design 91
Tools for Debugging 95
Putting It All Together 96
Powering Up 100
Add in the Processor 100
Some Thoughts on Debugging 100

Part II. Embedded Processors and Systems

5. **The PIC Microcontrollers** **107**
A Tale of Two Processors 107
Starting Simple 109
A Bigger PIC 113

6. **The AVR Microcontrollers** **115**
The AVR Architecture 116
The ATtiny15 Processor 118
Downloading Code 125
A Bigger AVR 127
Bus Interfacing 128

7. **68000-Series Computers** **156**
The 68000 Architecture 157
A Simple 68000-Based Computer 161

8. **DSP-Based Controllers** **170**
The DSP56800 173
A DSP56805-Based Computer 175
JTAG 183

Part III. Peripherals and Interfacing

9. Adding Peripherals Using SPI and I²C **187**
- Serial Peripheral Interface 187
- Inter Integrated Circuit 201
- Adding a Real-Time Clock with I²C 205
- Adding a Small Display with I²C 206

10. Serial Ports **208**
- UARTs 208
- Error Detection 210
- Old Faithful—RS-232C 211
- RS-422 218
- Infrared Communication 220
- USB 226

11. Networks **238**
- RS-485 238
- Controller Area Network (CAN) 241
- Ethernet 245

12. Analog **252**
- Amplifiers 252
- Analog-to-Digital Conversion 255
- Interfacing an External ADC 259
- Temperature Sensor 261
- Light Sensor 263
- Accelerometer 266
- Pressure Sensors 267
- Magnetic Field Sensor 270
- Digital-to-Analog Conversion 271
- PWM 273
- Motor Control 274
- Switching Big Loads 281

References **283**

Index **287**

Foreword

Embedded computers are the unsung heroes of modern life. An exercise I often set my undergraduate engineering students is to identify where they may have encountered embedded computers since waking up in the morning and arriving at their place of study or their place of work. Now, of course, there may be some who have older appliances around the house and drive an older car. Their embedded computer count may be fewer than 10—they probably have a compact disc player, and that already gets the count up and going. Think of any appliance sporting a non-basic user interface with buttons and a display, one that claims a better energy/water usage rating than the norm, one having to deal with digital data (CD players, for example), or one that communicates with other devices. Embedded hardware is behind it all. And that is just getting out of the door of the house. Think of the car, bus, or train to get to work. Think of the traffic control systems and the equipment used at work. This little exercise makes clear how embedded hardware outnumbers desktop PCs. In this book, John tells you how to design beasts such as these.

I have known and worked with John, as both an academic and an embedded systems engineer, for around 15 years now. I have seen him present university courses on embedded systems and design an assortment of embedded machines. John thoroughly enjoys working with students, imparting his knowledge and seeing students get things working. And the students enjoy it too. It is now great to see him capture even just a snippet of his expertise, enthusiasm, and experience in this book.

John has devoted much of his embedded computer development skills to wildlife research. He has built many dataloggers, all of which are compact and durable and have high data storage capacity and high operation lifetimes. Many albatrosses now fly around the southern oceans with machines designed by John attached to them. With these devices, scientists have learned a great deal more about the travels and feeding habits of these great birds. Albatrosses are nature's sleek and majestic flyers able to cruise long distances and with great precision. What better metaphor could there be for embedded computers?

In this book, John has walked the proverbial tightrope of taking the reader on a journey starting at the essentials and ending with a number of functional embedded computer designs. The journey is a pleasant and mentally stimulating one that provides just enough of everything, and the frequent anecdotes are ones to look forward to. And yet, at the conclusion of this book, one realizes that the embedded computer journey has only just begun. John's superb grounding opens the doors to the vast embedded universe.

Traditionally, books on electronics and microprocessors have assumed some high degree of competence in a broad range of topics. Typically readers are elevated in their knowledge; however, they often still fall short of being able to design a working system. Rather than taking a slice across the discipline as is traditionally done, John has taken a more streamed approach by walking the reader through a number of essential electronics topics. Each topic in its own right often has entire textbooks or courses devoted to it. I know John values the rigor with which such texts treat the various electronics components and systems, and certainly readers of this book are encouraged to bolster their knowledge from such sources; however, this book gives just enough to get going, and going a long way.

—Dr. Duncan A. Campbell

School of Electrical and Electronic Systems Engineering
Queensland University of Technology
Brisbane, Queensland, Australia

Preface

[Enlightenment] resides as comfortably in the circuits of a digital computer . . . as at the top of a mountain or in the petals of a flower.

—Robert M. Pirsig

Zen and the Art of Motorcycle Maintenance

This is a book about designing computer hardware and specifically about designing small machines for embedded applications. It is intentionally hardware specific. There are plenty of books out there on writing code for embedded systems (such as Michael Barr's excellent *Programming Embedded Systems in C and C++*, another O'Reilly & Associates title). What has been missing is a book that covers the nuts and bolts of developing embedded hardware. Sure, there are many books out there on microprocessors, but none that brings together all you need to create an embedded computer and make it go.

This is a book I have wanted to write for some time. It had its origins back in 1993, when I was lecturing at La Trobe University in Melbourne, Australia. I was given the task (at the last possible moment) of teaching a course in microprocessors to second-year students. The assigned text for the course was far from ideal. It talked about computer hardware but didn't show how to design computer hardware. It took a *Field of Dreams* approach—build it and it will go, with no consideration of timing, voltages, current draw, or anything else of importance. It was a newly published book, yet it covered components that had not been available for years. The memory chips it discussed were 128 bytes in capacity. (That's 128 *bytes*, not kilobytes!) This was a book that was neither relevant nor useful.

After talking to numerous representatives of publishing companies, I soon discovered that there wasn't much better available. And so, I solved the problem by writing detailed lecture notes for the students and told them to forget the textbook. These lecture notes were written quickly, and as a result, they were very rough indeed. The lectures were used to smooth the edges and fill the gaps. One day, I resolved, I would write a proper book.

And now, the opportunity has arisen to write for O'Reilly, and the book has become a reality. I no longer teach at La Trobe, having left many years ago to found my own company. More than ever, I want to bring together the real-world knowledge and experience necessary to construct working embedded systems. This book looks at the design process for creating and building embedded hardware and the analysis process for confirming that it will work. I will assume nothing about your knowledge beyond a rudimentary understanding of digital and analog electronics. The only real prerequisite is that you are intelligent and have an analytical mind. As I said at the start, this book is about hardware, and so you won't find software in these pages. That is better covered elsewhere.

Just as there is beauty in well-written software, there is beauty in well-designed hardware. With embedded computers, you get to understand the machine at all levels, at once aware of currents flowing through circuit traces and software executing complex algorithms. In fact, it is not possible to write embedded software without understanding the hardware, nor is it possible to design hardware without understanding software. You become involved with the machine to a degree beyond that which is possible with desktop computers. Best of all, it's a lot of fun.

In selecting chips and designs for this book, I have deliberately chosen parts that are both trivial to implement yet exceptionally useful. Aside from my own company (Embedded), I have no connection, financial or otherwise, with any of the companies or businesses mentioned in this book. You may, however, notice a prevalence of components from certain manufacturers. This simply reflects my personal preference for using their chips, based on my experience. Such companies produce chips that are easy to use, are reliable and robust, have great technical support, and provide thorough and comprehensive technical data. In other words, they have all the necessary prerequisites for inclusion in a book for beginners.

Many of the designs in this book look easy, and they are. They are intended as simple building blocks, allowing you to mix and match to achieve the embedded systems you need. There are some very complicated processors and support chips out there, and designs based upon them can be horrendously complicated, confusing, and frustrating. You won't find them in this book. This book is aimed at developing small, low-cost, and relatively simple embedded applications. I hope you will find it useful.

Organization of This Book

This book is divided into three parts. Part I covers fundamental concepts and introductory material. Part II looks at embedded processors and the design process for integrating them into systems. Part III looks at peripherals and adding functionality to your embedded systems.

Chapter 1 presents an overview of computer architectures and discusses the basics of an embedded system. Chapter 2 provides some background electronics theory and

introduces some important concepts. If you're already electronics-savvy, then you can skip on to Chapter 3, which covers providing power for your embedded system. In Chapter 4, you'll see how to physically produce and debug an embedded computer system. We'll also look at how to protect your embedded computer against electrical interference and other gremlins that can cause it grief.

Chapter 5 begins Part II of the book, where you'll encounter the first of the embedded processor architectures, the Microchip PIC. The PICs are tiny, self-contained computers that make building embedded systems easy and fun. Chapter 6 discusses the ATMEL AVR, another embedded processor ideally suited to small-scale, simple applications. You'll also learn how to add additional memory and peripherals to bus-based processors and discover the basics of memory management. With Chapter 7, we take a look at the Motorola 68000 series of processors. These chips have been around for quite some time and are still widely used. They are also a good starting point if you want to get into more complicated processors once you have more embedded experience. Chapter 8 examines processors based on Digital Signal Processing (DSP) architectures. These processors are adept at mathematically intensive and complex algorithms and are especially suited to control and sampling applications (such as the processing of digital signals).

In Part III of the book, you'll learn how to add function to your embedded computers by using peripherals. Chapter 9 covers SPI and I²C, two protocols that allow a wide range of small peripherals to be added to microcontrollers. Chapter 10 covers serial interfaces. These give your embedded system access to host computers and to external peripherals such as modems. We'll also take a look at RS-232C, RS-422, infrared communication, and USB. Networks are covered in Chapter 11, where you'll see how to add two low-cost industrial networks (RS-485 and CAN) to your embedded computer. Also in Chapter 11, you'll learn how to add an Ethernet port to your embedded system, by which you can connect to other computers, servers, and gateways and, through them, to the Internet. Finally, Chapter 12 looks at real-world interfacing. You'll learn how to convert analog signals into digital values for processing and, conversely, how to convert digital values back into analog voltages. You'll learn how to measure temperature, light, pressure, acceleration, and magnetic fields in your embedded system using sensors, as well as how to use an embedded computer to control small electric motors.

Acknowledgments

In the past, I have often read in prefaces how authors thank their editors for the help they gave. It is only now that I understand the depth and significance of this help. I'd like to give a special thank you to my editor, Dr. Jon Orwant. This book is a significantly better work thanks to his good humor, insightful comments, and brilliance.

As you have no doubt already noticed, O'Reilly publishes beautifully presented books. I would like to thank the production team, Lorrie LeJeune, Tatiana Diaz,

Larry Sweazy, Jessamyn Read, Rob Romano, Norma Emory, Laura Gabler, Emma Colby, Mike Sierra, Ellie Volckhausen, David Futato, and Philip Dangler for their hard work. They have contributed as much to the readability of this book as I have in writing it.

I'd like to thank Dr. Duncan Campbell for his friendship, encouragement, and assistance in proofreading this work. Duncan's camaraderie and professional support have meant a lot.

Geoff McDonald has been a great friend and has made many helpful suggestions regarding the content of this book. He has also proofread, and I thank him for all his help.

I'd like to thank Dr. Jeff O'Keefe for his long friendship and support over the years. He's been a good friend ever since we were undergrads together, blowing up integrated circuits and irradiating lecturers in second-year lab!

Thanks to Michael Barr, John Redford, and John Watlington for reviewing the "prototype" of this book and offering several useful suggestions.

Thanks to Professor Anthony Maeder and the staff of the School of Electrical and Electronic Systems Engineering at Queensland University of Technology for their assistance.

Thanks also to the staff at Agora for logistical support. You've all been a great help.

I'd like to thank my friends and colleagues Michael Lees, David Nicholls, Peter Stewart, Mark Gentile, Professor John Devlin, Richard Wiltshire, John Williams, Michelle and Robert Salier, and Dr. Peter O'Shea. Thanks also to Louisa Sciacca and David Kerven for providing ideas for some of the examples in this book.

Finally and most importantly, I'd like to thank my extended family for their love and support. Most especially, I'd like to thank my sister, Kris, her husband, Duncan, and my two nephews, Andrew and James, whose love and good humor have made life worth living. I'd also like to thank Chris and Jeff Goopy for always being there and my cousins, Theo and Maree, David and Jenevieve, Michael, Andrew and Karen, Antony, and Fiona, for their friendship and support. A special thank you to my uncles, Vince and Dave Catsoulis, who have shown me the meaning of love, honor, and strength of character. I owe them much.

Online Resources

The latest technical data on devices should be obtained directly from the component manufacturers' web sites. For less expensive components, some manufacturers (such as Analog Devices and Maxim) allow you to order free samples directly via their web pages.

The following URLs may be useful:

http://www.agere.com	Agere Systems
http://www.agilent.com	Agilent Technologies
http://www.altera.com	Altera (programmable logic)
http://www.analogdevices.com	Analog Devices
http://www.atmel.com	ATMEL
http://www.cirrus.com	Cirrus Logic
http://www.embedded.com	Embedded Systems magazine
http://www.gnu.org	The GNU Free Software Foundation
http://www.htsoft.com	Hitech (commercial C compilers)
http://www.irf.com	International Rectifier
http://www.matrixorbital.com	Matrix Orbital (displays)
http://www.maxim-ic.com	Maxim
http://www.microchip.com	Microchip
http://e-www.motorola.com	Motorola (Semiconductor Division)
http://www.mskennedy.com	M. S. Kennedy (motor control)
http://www.national.com	National Semiconductor
http://www.st.com	ST Electronics
http://www.taosinc.com	Texas Advanced Optical Sensors (TAOS)
http://www.ti.com	Texas Instruments
http://www.vishay.com	Vishay (optoelectronics)
http://www.winbond.com	Winbond (peripherals)
http://www.xicor.com	Xicor (nonvolatile memory)
http://www.xilinx.com	Xilinx (programmable logic)

Conventions

The conventions used in this book are as follows:

Main text

```
Source Code
```

Signal (high active)

$\overline{\textbf{Signal}}$ (low active)

Hexadecimal numbers in this book are denoted with the prefix *0x*.

Binary numbers are denoted by the prefix %.

K is 1024, while *k* is 1000.

Disclaimer

Much of the information contained in this book is based on personal knowledge and experience. While I believe that the information contained herein is correct, I accept no responsibility for its validity. The hardware designs, software, and descriptive text contained herein are provided for educational purposes only. It is the responsibility of the reader to independently verify all information. Original manufacturers' data should be used at all times when implementing a design.

The author, Embedded Pty. Ltd., and O'Reilly & Associates, Inc., make no warranty, representation, or guarantee regarding the suitability of any hardware or software described herein for any particular purpose, nor do they assume any liability arising out of the application or use of any product, system, circuit, or software and specifically disclaim any and all liability, including, without limitation, consequential or incidental damages. The hardware and software described herein are not designed, intended, nor authorized for use in any application intended to support or sustain life or any other application in which the failure of a system could create a situation in which personal injury, death, loss of data or information, or damages to property may occur. Should the reader implement any design described herein for any application, the reader shall indemnify and hold the author, O'Reilly & Associates, Inc., Embedded Pty. Ltd., and their respective shareholders, officers, employees, and distributors harmless against all claims, costs, damages and expenses, and reasonable solicitor fees arising out of, directly or indirectly, any claim of personal injury, death, loss of data or information, or damages to property associated with such unintended or unauthorized use.

—John Catsoulis
Brisbane, Australia
October 2002
jtc@embedded.com.au
http://www.embedded.com.au

PART I
Background

I introduce the basic concepts of computer architecture in Chapter 1 and then cover some introductory electronics theory in Chapter 2. In Chapter 3, we'll look at powering your embedded designs, and in Chapter 4, construction and fabrication techniques are discussed.

CHAPTER 1

Introduction to Computer Architecture

Each machine has its own, unique personality which probably could be defined as the intuitive sum total of everything you know and feel about it. This personality constantly changes, usually for the worse, but sometimes surprisingly for the better . . .

—Robert M. Pirsig

Zen and the Art of Motorcycle Maintenance

This book is about designing and building specialized computers. We all know what a computer is. It's that box that sits on your desk, quietly purring away (or rattling if the fan is shot), running your programs and regularly crashing (if you're not running some variety of Unix). Inside that box is the electronics that runs your software, stores your information, and connects you to the world. It's all about processing information. Designing a computer, therefore, is about designing a machine that holds and manipulates data.

Computer systems fall into essentially two separate categories. The first, and most obvious, is that of the desktop computer. When you say "computer" to someone, this is the machine that usually comes to his mind. The second type of computer is the embedded computer, a computer that is integrated into another system for the purposes of control and/or monitoring. Embedded computers are far more numerous than desktop systems, but far less obvious. Ask the average person how many computers she has in her home, and she might reply that she has one or two. In fact, she may have 30 or more, hidden inside her TVs, VCRs, DVD players, remote controls, cell phones, ovens, toys, and a host of other devices. In this chapter, we'll look at computer architecture in general, which applies to both embedded and desktop computers.

The underlying architectures of desktop computers and embedded computers are fundamentally the same. At a crude level, both have a processor, memory, and some form of input and output. The primary difference lies in their intended use, and this is reflected in their software. Desktop computers can run a variety of application programs, with system resources orchestrated by an operating system. By running

different application programs, the functionality of the desktop computer is changed. One moment, it may be used as a word processor; the next, it is an MP3 player or a database client. Which software is loaded and run is under user control.

In contrast, the embedded computer is normally dedicated to a specific task. The advantage of using an embedded microprocessor over dedicated electronics is that the functionality of the system is determined by the software, not the hardware. It typically has one application and one application only, and this is permanently running. The embedded computer may or may not have an operating system, and rarely does it provide the user with the ability to arbitrarily install new software. The software is normally contained in the system's nonvolatile memory, unlike a desktop computer in which the nonvolatile memory contains boot software and (maybe) low-level drivers only.

Embedded hardware is often much simpler than a desktop system, but it can also be far more complex too. An embedded computer may be implemented in a single chip with just a few support components, and its purpose may be as crude as a controller for a garden-watering system. Or the embedded computer may be a 150-processor, distributed parallel machine responsible for all the flight and control systems of a commercial jet. As diverse as embedded hardware may be, the underlying principles of design are the same.

This chapter introduces some *important* concepts relating to computer architecture, with specific emphasis on those topics relevant to embedded systems. Its purpose is to give you grounding before moving on to the more hands-on information that begins in Chapter 2. In this chapter, you'll learn about the basics of processors, interrupts, the difference between RISC and CISC, parallel systems, memory, and I/O.

Concepts

At the simplest level, a computer is a machine designed to process, store, and retrieve data. Data may be numbers in a spreadsheet, characters of text in a document, dots of color in an image, waveforms of sound, or the state of some system, such as an air conditioner or a CD player. It is important to note that all data is stored in the computer as numbers.

The computer manipulates the data by performing operations on the numbers. Displaying an image on a screen is accomplished by moving an array of numbers to the video memory, each number representing a pixel of color. To play an MP3 audio file, the computer reads an array of numbers from disk and into memory, manipulates those numbers to convert the compressed audio data into raw audio data, and then outputs the new set of numbers (the raw audio data) to the audio chip.

Everything that a computer does, from web browsing to printing, involves moving and processing numbers. The electronics of a computer is nothing more than a system designed to hold, move, and change numbers.

A computer system is composed of many parts, both hardware and software. At the heart of the computer is the processor, the hardware that executes the computer programs. The computer also has memory, often several different types in the one system. The memory is used to store programs while the processor is running them, as well as to store data that the programs are manipulating. The computer also has devices for storing data or exchanging data with the outside world. These may allow the input of text via a keyboard, the display of information on a screen, or the movement of programs and data to or from a disk drive.

The software controls the operation and functionality of the computer. There are many "layers" of software in the computer (Figure 1-1). Typically, a given layer will interact with only the layer immediately above or below.

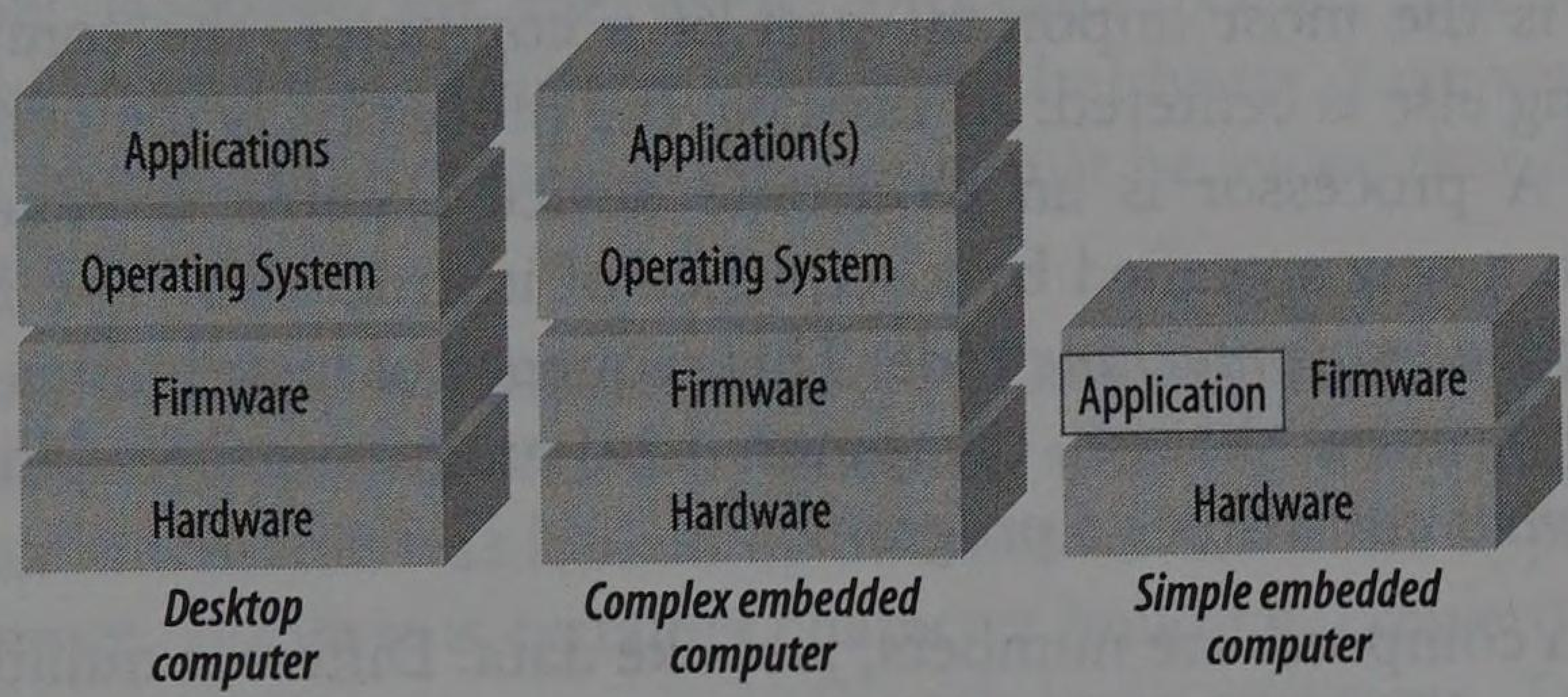


Figure 1-1. Software layers

At the lowest level are programs that are run by the processor when the computer first powers up. These programs initialize the other hardware subsystems to a known state and configure the computer for correct operation. This software, because it is permanently stored in the computer's memory, is known as *firmware*.

The *bootloader* is located in the firmware. The bootloader is a special program run by the processor that reads the operating system from disk (or nonvolatile memory or network) and places it in memory so that the processor may then run it. The bootloader is present in desktop computers and workstations and may also be present in some embedded computers.

Above the firmware, the operating system controls the operation of the computer. It organizes the use of memory; controls devices such as the keyboard, mouse, screen, disk drives; and so on. It is also the software that often provides an interface to the user, enabling him to run application programs and access his files on disk. The operating system also provides a set of software tools for application programs, providing a mechanism by which they too can access the screen, disk drives, and so on. Not all embedded systems use or even need an operating system. Often, an embedded system will simply run code dedicated to its task, and the presence of an operating system is overkill. In other instances, such as network routers, an operating system provides necessary software integration and greatly simplifies the development

process. Whether an operating system is needed and useful really depends on the intended purpose of the embedded computer and, to a lesser degree, on the preference of the designer.

At the highest level, the *application software* constitutes the programs that provide the functionality of the computer. Everything below the application is considered *system software*. For embedded computers, the boundary between application and system software is often blurred. This reflects the underlying principle in embedded design that a system should be designed to achieve its objective in as simple and straightforward a manner as possible.

Processors

The processor is the most important part of a computer, the component around which everything else is centered. In essence, the processor is the computing part of the computer. A processor is an electronic device capable of manipulating data (information) in a way specified by a sequence of instructions. The instructions are also known as *opcodes* or *machine code*. This sequence of instructions may be altered to suit the application; hence, computers are programmable. The sequence of instructions is what constitutes a program.

Instructions in a computer are numbers, just like data. Different numbers, when read and executed by a processor, cause different things to happen. A good analogy is the mechanism of a music box. A music box has a rotating drum with little bumps and a row of prongs. As the drum rotates, different prongs in turn are activated by the bumps, and music is produced. In a similar way, the bit patterns of instructions feed into the execution unit of the processor. Different bit patterns activate or deactivate different parts of the processing core. Thus, the bit pattern of a given instruction may activate an addition operation, while another bit pattern may cause a byte to be stored to memory.

A sequence of instructions is a machine-code program. Each type of processor has a different *instruction set*, meaning that the functionality of the instructions (and the bit patterns that activate them) vary. Processor instructions are often quite simple, such as "add two numbers" or "call this function." In some processors, however, they can be as complex and sophisticated as "if the result of the last operation was zero, then use this particular number to reference another number in memory, and then increment the first number once you've finished." This will be covered in more detail in the section on CISC and RISC processors, later in this chapter.

A program that a given processor may execute might look something like:

```
B0 4F F7 01 00 07...
```

Humans find such programs very hard to write and even harder to understand. To make this easier for us to use, we use a notation called *assembly language*, in which

mnemonics are used to represent the opcodes. Assembly language instructions equate directly to their machine-code counterparts.

For example, the instruction `B04FF7` is more easily understood by its assembly language mnemonic `ADD.B #0xFF, W7`. This is still a bit cryptic, so we usually add comments on the righthand side to help us follow what is going on.

So, the preceding machine code written in assembly would be:

```
Assembly              Comments
ADD.B #0xFF, W7       ; Add the byte -1 to register W7
CALL W7               ; call the subroutine pointed to by W7
```

Different processor families use different assembly languages. No two are alike, although some degree of similarity may be present. The previous examples are written in assembly language for the dsPIC processor. Other assembly languages, because they are based on very different processor hardware, have very different syntax. This is not of great importance to this book; just be aware that different processors use very different code.

No computer can understand assembly directly. Back in the olden days, when computers were steam-driven and tended by gnomes, software was compiled manually. Each instruction mnemonic was looked up and converted to the appropriate opcode by the programmer. While it is certainly character building, converting from assembly to opcodes is very tiresome, particularly with large programs. To make life easier, special compilers, called *assemblers*, take mnemonics and convert them to opcodes.

Assembly language has been described as the "nuts-and-bolts language," for you are writing code directly for the processor. For a lot of the software you will write, a high-level language like C will be the language of choice. High-level languages make developing software much easier, and your code is also portable (to a degree) between different target machines. Compilers of high-level languages convert your source code down to machine opcodes. Thus, by using a compiler, the programmer is relieved of having to know the specific details of the processor and of having to code her program directly in machine code.

So there are good reasons for using a high-level language. Yet, many times programmers write directly in assembly language. Why? Assembly and machine code, because they are "handwritten," can be finely tuned to get the most performance out of the processor and computer hardware. This can be particularly important when dealing with time-critical operations with I/O devices. Further, coding directly in assembly can sometimes (but not always) result in a smaller code space. So, if you're trying to cram complex software into a small amount of memory and need that software to execute quickly and efficiently, assembly language may be your best (and only) choice. The drawback, of course, is that the software is harder to maintain and has zero portability to other processors. A good programmer can create more efficient code than the average C compiler; however, a good C compiler will probably

produce tighter code than a mediocre programmer. Typically, you can include *inline assembly* within your C code and thereby get the best of both worlds.

At the mere mention of assembly language, many a die-hard programmer begins to quiver in fear, as if just invited into a tiger's cage. But assembly-language programming is not that hard and can often be a lot of fun. Think of it as being "as one" with the processor.

That said, this is a book about hardware, not software. Embedded software development is already covered by two O'Reilly & Associates books: *Programming Embedded Systems in C and C++*, by Michael Barr, and *Programming with GNU Software*, by Mike Loukides and Andy Oram.

When you're developing your embedded system, it is best to start with a *development kit* from the processor's manufacturer. A good development kit will not only provide you with a working example of the machine you're trying to build (and upon which you can test your code), it should also include a nice *Integrated Development Environment* (or *IDE*). The IDE will have a windowing editor, a debugger, a simulator too if you're lucky, an assembler, and hopefully a C compiler as well. The kit should also come with cables and tools for programming the processor and circuit schematics so you can see what a working machine should look like. Treat the schematics with a small degree of caution. Some (but not all) semiconductor manufacturers farm out the design of their development systems to small, external companies. Some of these companies do a fantastic job, while others seem to employ stray chimpanzees as design engineers. In the latter case, the development system will work, but only through a miracle and by the grace of the digital gods. So, treat the schematics as a rough guide only.

To use the IDE, you will need a desktop computer. And here's the bad news. Almost without exception, the IDEs will run on only one platform and under only one operating system. No prizes for guessing which one. So, if your preferred environment is a Unix workstation, generally you're out of luck. While the GNU tools are great, sometimes you just have to resort to the IDE to download code into your target computer, particularly for 8- and 16-bit processors.

Development kit prices range from free (if you're at the right place at the right time) to many tens of thousands of dollars for some of the really high-end and exotic processors. For most embedded-type processors, you could expect to pay somewhere between \$50 and \$300, depending on the chip, the manufacturer, and its current whim. The time a development kit will save you probably makes the investment worthwhile.

System Architecture

The processor alone is incapable of successfully performing any tasks. It requires memory (for program and data storage), support logic, and at least one I/O device

(input/output device) used to transfer data between the computer and the outside world. The basic computer system is shown in Figure 1-2.

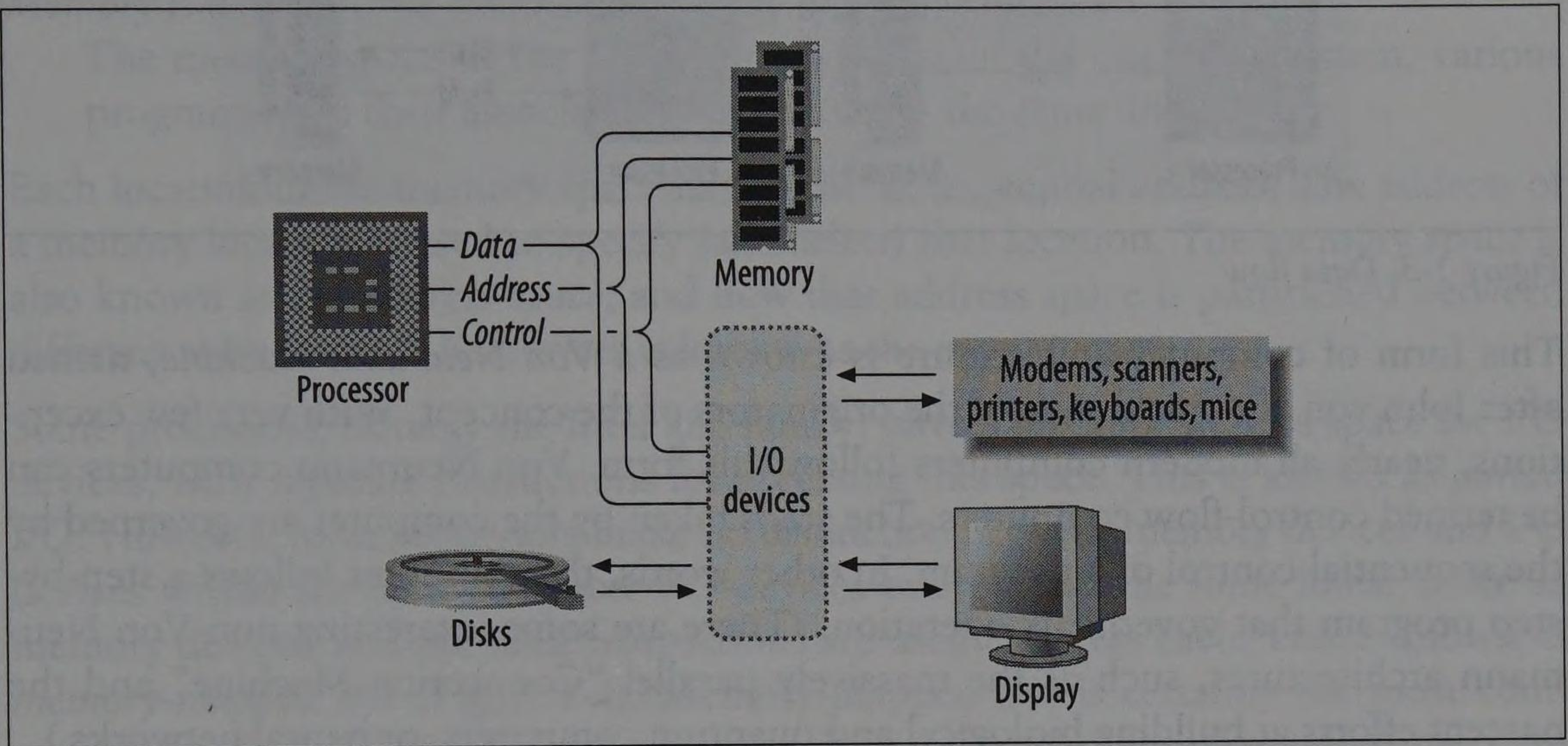


Figure 1-2. Basic computer system

A *microprocessor* is a processor implemented (usually) on a single, integrated circuit. With the exception of those found in some large supercomputers, nearly all modern processors are microprocessors, and the two terms are often used interchangeably. Common microprocessors in use today are the Intel Pentium series, Motorola/IBM PowerPC, MIPS, ARM, and Sun SPARC. A microprocessor is sometimes also known as a *CPU* (*Central Processing Unit*).

A *microcontroller* is a processor, memory, and some I/O contained within a single, integrated circuit and intended for use in embedded systems. The buses that interconnect the processor with its I/O exist within the same integrated circuit. The range of available microcontrollers is very broad. They range from the tiny PICs and AVRs (to be covered in this book), to PowerPC processors with built-in I/O, intended for embedded applications.

Microcontrollers are very similar to *System-On-Chip* (*SOC*) processors, intended for use in conventional computers such as PCs and workstations. SOC processors have a different suite of I/O, reflecting their intended application, and are designed to be interfaced to large banks of external memory. Microcontrollers usually have all their memory on-chip and may provide only limited support for external memory devices.

The memory of the computer system contains both the instructions that the processor will execute and the data it will manipulate. The memory of a computer system is never empty. It always contains something, whether it be instructions, meaningful data, or just the random garbage that appeared in the memory when the system powered up.

Instructions are read (fetched) from memory, while data is both read from and written to memory, as shown in Figure 1-3.

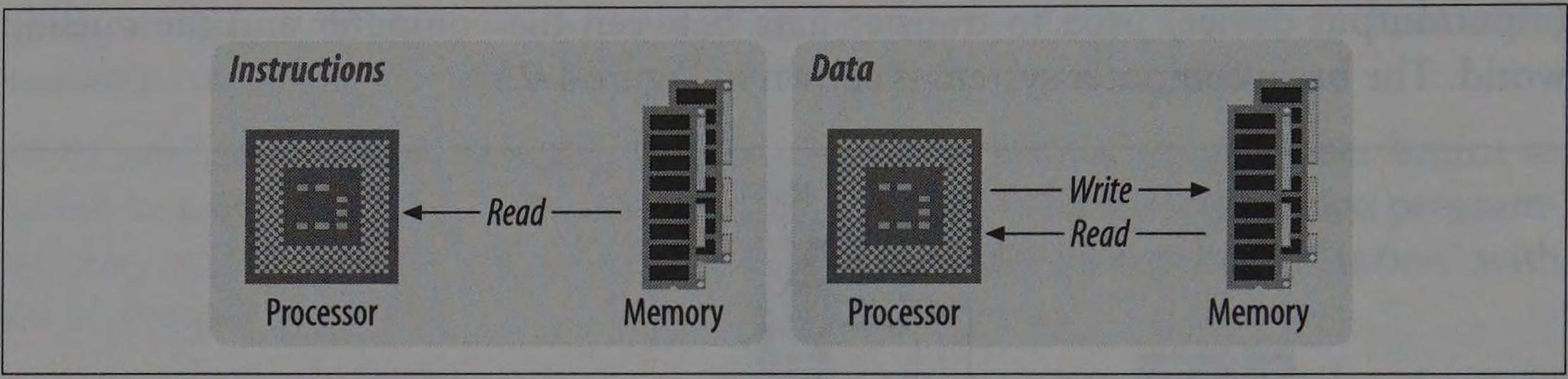


Figure 1-3. Data flow

This form of computer architecture is known as a *Von Neumann machine*, named after John von Neumann, one of the originators of the concept. With very few exceptions, nearly all modern computers follow this form. Von Neumann computers can be termed control-flow computers. The steps taken by the computer are governed by the sequential control of a program. In other words, the computer follows a step-by-step program that governs its operation. (There are some interesting non-Von Neumann architectures, such as the massively parallel "Connection Machine" and the nascent efforts at building biological and quantum computers, or neural networks.)

A classical Von Neumann machine has several distinguishing characteristics:

There is no real difference between data and instructions.
: A processor can be directed to begin execution at a given point in memory, and it has no way of knowing whether the sequence of numbers beginning at that point is data or instructions. The instruction `0x4143` may also be data (the number 0x4143 or the ASCII characters "A" and "C"). The processor has no way of telling what is data or what is an instruction. If a number is to be executed by the processor, it is an instruction; if it is to be manipulated, it is data.

  Because of this lack of distinction, the processor is capable of changing its instructions (treating them as data) under program control. And because the processor has no way of distinguishing between data and instruction, it will blindly execute anything that it is given, whether it is a meaningful sequence of instructions or not.

Data has no inherent meaning.
: There is nothing to distinguish between a number that represents a dot of color in an image and a number that represents a character in a text document. Meaning comes from how those numbers are treated under the execution of a program.

Data and instructions share the same memory.
: This means that sequences of instructions in a program may be treated as data by another program. A compiler creates a program binary by generating a sequence of numbers (instructions) in memory. To the compiler, the compiled program is just data, and it is treated as such. It is a program only when the processor begins execution. Similarly, an operating system loading an application program from disk does so by treating the sequence of instructions of that pro-

gram as data. The program is loaded to memory just as an image or text file would be, and this is possible due to the shared memory space.

Memory is a linear (one-dimensional) array of storage locations.
The memory space of the processor may contain the operating system, various programs, and their associated data, all within the same linear space.

Each location in the memory space has a unique, sequential address. The address of a memory location is used to specify (and select) that location. The memory space is also known as the *address space*, and how that address space is partitioned between different memory and I/O devices is known as the *memory map*.

Some processors, notably the Intel x86 family, have a separate address space for I/O devices, with separate instructions for accessing this space. This is known as *ported I/O*. However, most processors make no distinction between memory devices and I/O devices within the address space. I/O devices exist within the same linear space as memory devices, and the same instructions are used to access each. This is known as *memory-mapped I/O* (Figure 1-4). Memory-mapped I/O is certainly the most common form. Ported I/O address spaces are becoming rare, and the use of the term even rarer.

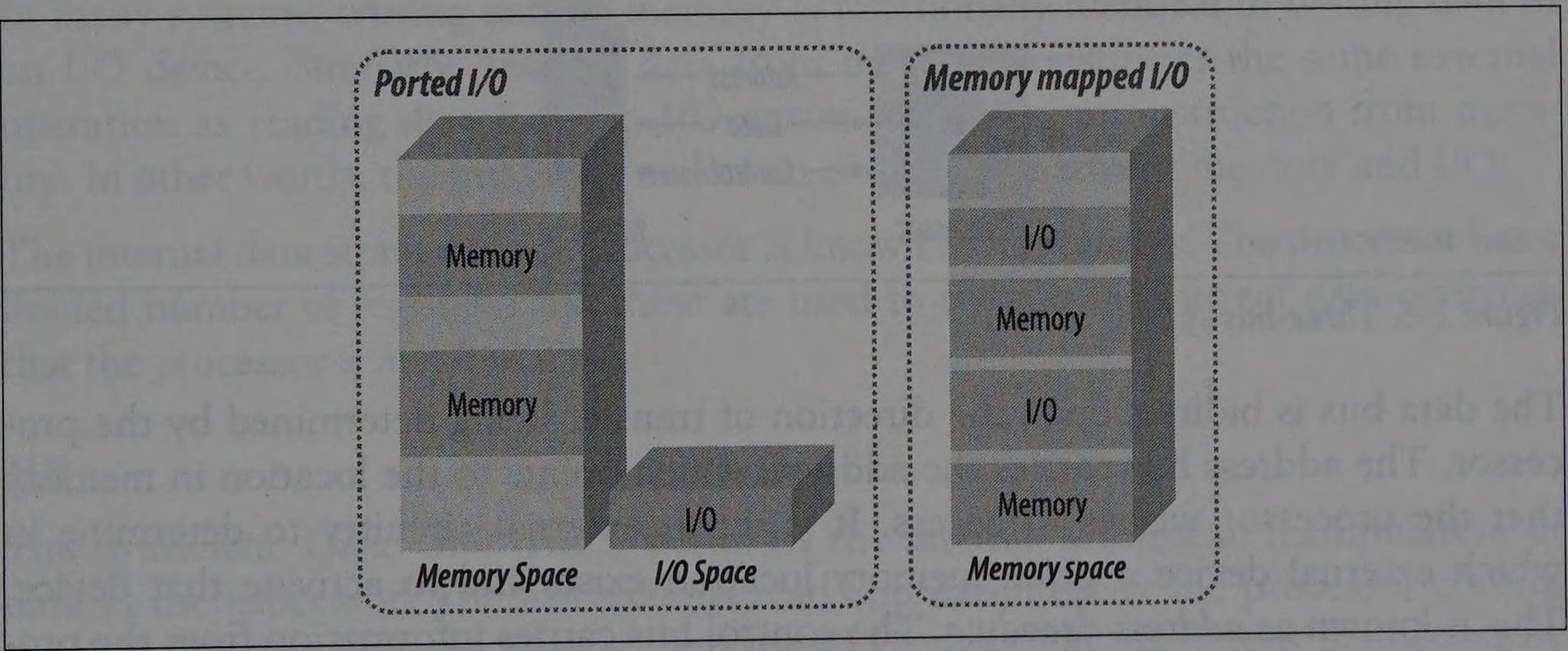


Figure 1-4. Ported versus memory-mapped I/O spaces

Most microprocessors available are standard Von Neumann machines. The main deviation from this is the *Harvard architecture*, in which instructions and data have different memory spaces (Figure 1-5), with separate address, data, and control buses for each memory space. This has a number of advantages in that instruction and data fetches can occur concurrently, and the size of an instruction is not set by the size of the standard data unit (word).

Buses

A bus is a physical group of signal lines that have a related function. Buses allow for the transfer of electrical signals between different parts of the computer system and

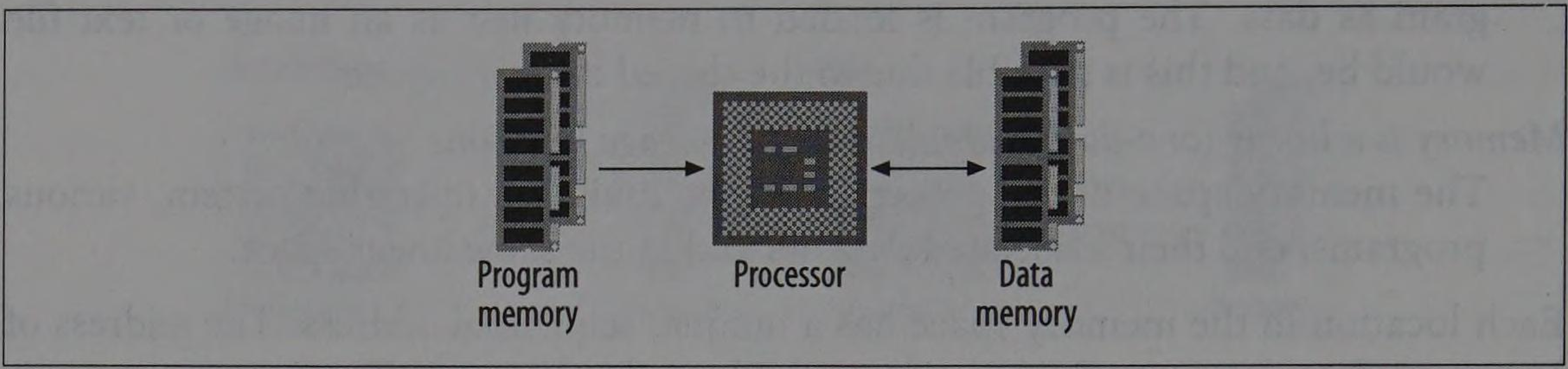


Figure 1-5. Harvard architecture

thereby transfer information from one device to another. For example, the data bus is the group of signal lines that carry data between the processor and the various subsystems that constitute the computer. The width of a bus is the number of signal lines dedicated to transferring information. For example, an 8-bit-wide bus transfers 8 bits of data in parallel.

The majority of microprocessors available today (with some exceptions) use the three-bus system architecture (Figure 1-6). The three buses are the *address bus*, the *data bus*, and the *control bus*.

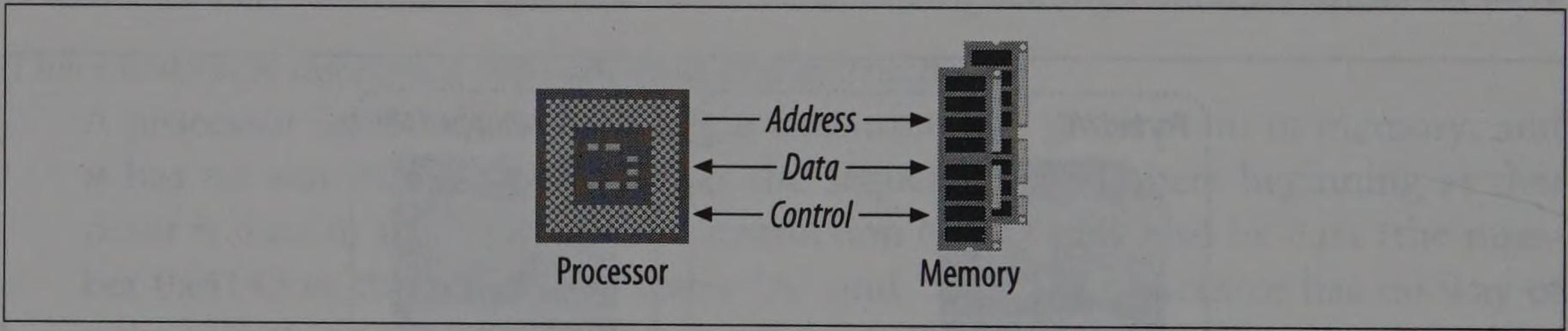


Figure 1-6. Three-bus system

The data bus is bidirectional, the direction of transfer being determined by the processor. The address bus carries the address, which points to the location in memory that the processor wishes to access. It is up to external circuitry to determine in which external device a given memory location exists and to activate that device. This is known as *address decoding*. The control bus carries information from the processor about the state of the current access, such as whether it is a write or a read operation. The control bus can also carry information back to the processor regarding the current access, such as an address error. Different processors have different control lines, but some control lines are common among many processors. The control bus may consist of output signals such as read, write, valid address, and so on. A processor has several input control lines too, such as RESET, one or more interrupt lines, and a clock input.

A few years ago, I had the opportunity to wander through, in, and around CSIRAC (pronounced "sigh-rack"). This was one of the world's first digital computers, designed and built in Sydney, Australia, in the late 1940s. It was a massive machine, filling a very big room with the type of solid hardware that you can really kick. It was quite an experience looking over the old machine. I remember at one stage walking *through* the disk controller (it was the size of a small room) and looking up at a mass of wires strung overhead. I asked what they were for. "That's the data bus!" came the reply.

CSIRAC is now housed in the museum of the University of Melbourne. You can take an online tour of the machine, and even download a simulator, at *http://www.cs.mu.oz.au/csirac.*

Processor operation

There are six basic functions that a processor can perform. The processor can write data to system memory or write data to an I/O device; it can read data from system memory or read data from an I/O device; it can read instructions from system memory; and it can perform internal manipulation of data within the processor.

In many systems, writing data to memory is functionally identical to writing data to an I/O device. Similarly, reading data from memory constitutes the same external operation as reading data from an I/O device or reading an instruction from memory. In other words, the processor makes no distinction between memory and I/O.

The internal data storage of the processor is known as its *registers*. The processor has a limited number of registers, and these are used to contain the current data/operands that the processor is manipulating.

ALU

The *Arithmetic Logic Unit (ALU)* performs the internal arithmetic manipulation of data in the processor. The instructions read and executed by the processor control the data flow between the registers and the ALU, as well as operations performed by the ALU, via the ALU's control inputs. A symbolic representation of an ALU is shown in Figure 1-7.

Whenever instructed by the processor, the ALU performs an operation (typically one of addition, subtraction, multiplication, division, NOT, AND, NAND, OR, NOR, XOR, shift left/right, or rotate left/right) on one or more values. These values, called operands, are typically obtained from two registers or from one register and a memory location. The result of the operation is then placed back into a given destination register or memory location. The status outputs indicate any special attributes about the operation, such as whether the result was zero or negative or if an overflow or carry occurred. Some processors have separate units for multiplication and division and for bit shifting, providing faster operation and increased throughput.

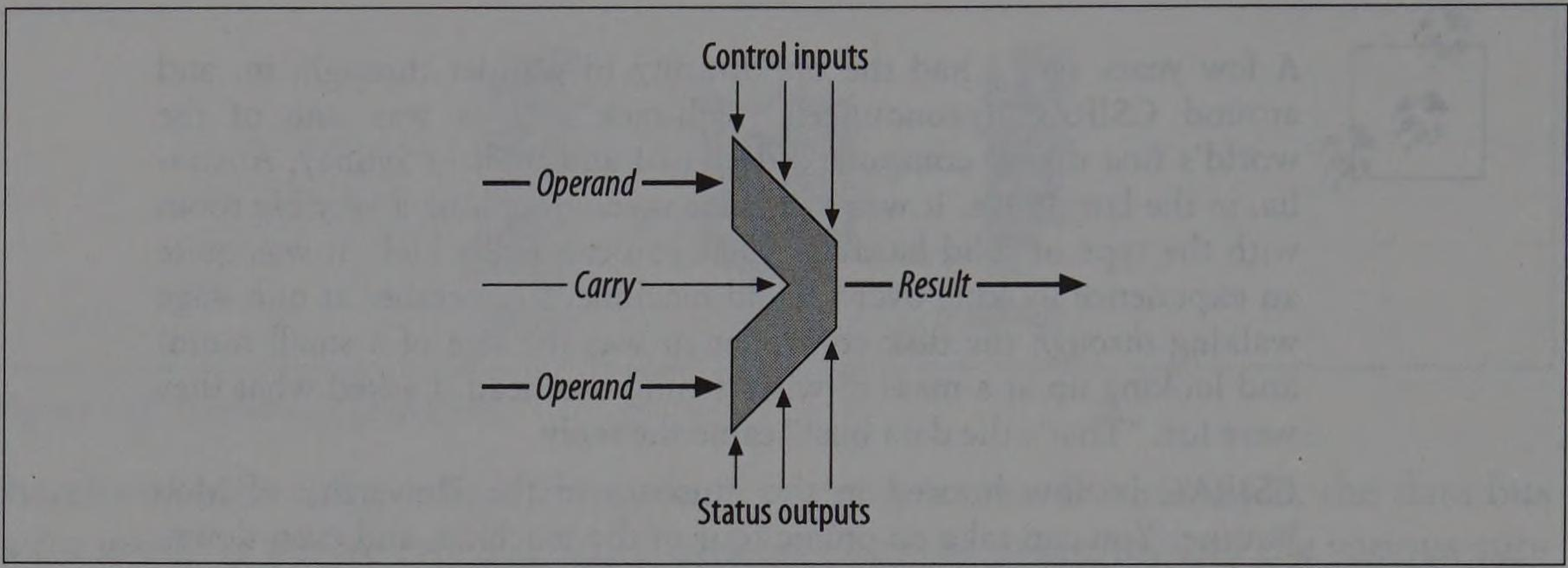


Figure 1-7. ALU block diagram

Each architecture has its own unique ALU features, which can vary greatly from one processor to another. However, all are just variations on a theme and all share the common characteristics just described.

Registers

Registers are the internal (working) storage for the processor. The number of registers varies significantly between processor architectures. Typically, the processor will have one or more *accumulators*. These are registers that may have arithmetic operations performed upon them. In some architectures, all the registers function as accumulators, whereas in others, some registers are dedicated for storage only and have limited functionality.

Some processors have *index registers* that can function as pointers into the memory space. In some architectures, all general-purpose registers can act as index registers; in others, dedicated index registers exist.

All processors will have a *program counter* (also known as an *instruction pointer*) that tracks the location in memory of the next instruction to be fetched and executed. All processors have a *status register* (also known as a *condition-code register*, or *CCR*) that consists of various status bits (flags) that reflect the current operational state. Such flags might indicate whether the result of the last operation was zero or negative, whether a carry occurred, if an interrupt is being serviced, and so on.

Some processors also have one or more *control registers*, consisting of configuration bits that affect processor operation and the operating modes of various internal subsystems. Many peripherals also have registers that control their operation and registers that contain the results of operations. These peripheral registers are normally mapped into the address space of the processor.

Some processors have banks of *shadow registers*, which save the state of the main registers when the processor begins servicing an interrupt (to be discussed shortly).

Processors are commonly 8-bit, 16-bit, 32-bit, or 64-bit, referring to the width of their registers. An 8-bit processor is invariably low-cost and is suitable for relatively simple control and monitoring applications. If more processing power is required, the larger processors are preferable, although cost and system complexity go up accordingly.

Stacks

Many processors implement one or more *stacks*, which serve as temporary storage in external memory. The processor can *push* a value from a register on the stack to preserve it for later use. The processor retrieves this value by *popping* from the stack back into a register. In some processor architectures, popping is also known as *pulling*.

Most processors have a *stack pointer*, which references the next free location on the stack. Some processors implement more than one stack and so have more than one stack pointer. Most stacks grow down through memory. (Some processors have stacks that grow up as the stack is filled.) When the processor pushes or pops a value to or from the stack, the stack pointer automatically decrements (or increments) to point to the next free location.

Addressing modes

The different ways in which an instruction can reference a register or memory location are known as the *addressing modes* of the processor. The types of addressing modes available within different architectures vary, but the basic ones are as follows:

Inherent
: The instruction deals purely with registers.

Immediate/literal
: The instruction has a literal number as an operand.

Direct
: The instruction accesses a memory location, specified by a short address. In other words, direct addressing provides access to a subset of the total address space. On a processor with a 16-bit address bus, a direct access would specify an address within the first 256 bytes. On a 32-bit processor, a direct access may specify an address within the first 64K of memory, for example. Direct addressing is used (when possible) to reduce the length of instructions referencing memory. This can reduce code size and therefore instruction fetch time in time-critical applications.

Extended
: The instruction accesses a memory location, specified by the full address.

Indexed
: The instruction uses the contents of a register as a pointer into memory.

Relative
An offset is specified as part of the addressing. For example, a branch instruction uses relative addressing to add (or subtract) a value from the program counter.

Big-endian and little-endian

Microprocessors are either *big endian* or *little endian* in their architecture. This refers to the way in which the processor stores data (16 bits or greater) to memory. A big-endian processor stores the most significant byte at the least significant address, as illustrated in Figure 1-8. In each case, the data has been stored to address 0x0100.

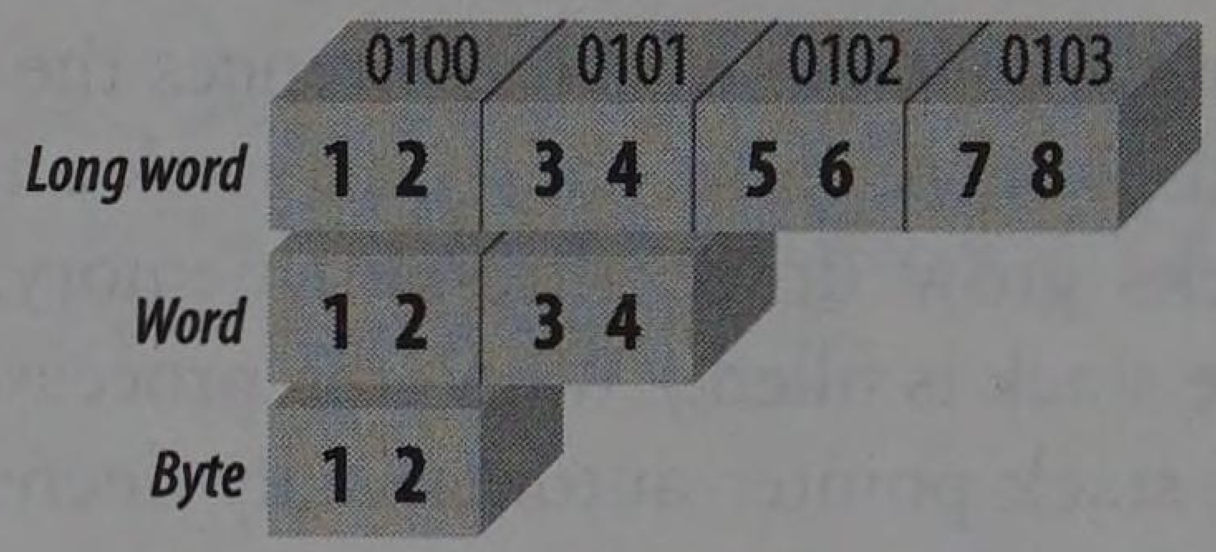


Figure 1-8. Big endian

A little-endian processor stores the most significant byte at the most significant address, as shown in Figure 1-9.

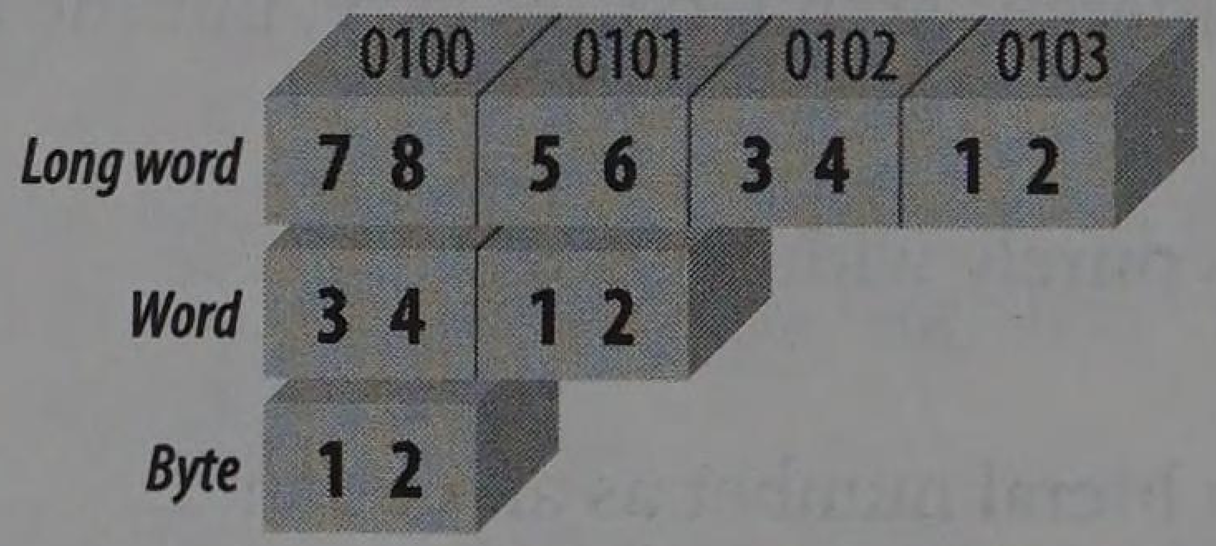


Figure 1-9. Little endian

With the little-endian scheme, the least significant data travels over the least significant part of the data bus and is stored at the least significant memory location. In other words, for a programmer, it is conceptually easier to understand in terms of data path. The disadvantage of little endian is that data appears backward in the computer's memory. Storing the value 0x12345678 to memory results in 0x78563412 in the memory space. Note that a little-endian processor will read this data back correctly; it's just that it makes it harder to understand the numbers if a human is looking at the memory directly. Alternatively, a big-endian processor storing 0x12345678 to memory results in 0x12345678 sitting inside the memory chip. This appears (to a human) to make more sense. Neither scheme has much advantage over the other in terms of operation; they are just two different ways of doing the same thing. When you're doing high-level programming on a system, the "endianness" makes little difference, for you are rarely exposed to it. However, when you are

developing and debugging hardware and low-level firmware, you come across it all the time, so an understanding of big endian and little endian is important.

Interrupts

Interrupts (also known as *traps* or *exceptions* in some processors) are a technique of diverting the processor from the execution of the current program so that it may deal with some event that has occurred. Such an event may be an error from a peripheral or simply that an I/O device has finished the last task it was given and is now ready for another. An interrupt is generated in your computer every time you press a key or move the mouse. Interrupts alleviate the processor from having to continuously check the I/O devices to determine whether they require service. Instead, the processor may continue with other tasks. The I/O devices will notify it if and when they require attention by asserting one of the processor's interrupt inputs. Interrupts can be of varying priorities in some processors, thereby assigning differing importance to the events that can interrupt the processor. If the processor is servicing a low-priority interrupt, it will pause that in order to service a higher-priority interrupt. However, if the processor is servicing an interrupt and a second, lower-priority interrupt occurs, the processor will ignore that interrupt until it has finished the higher-priority service.

When an interrupt occurs, the processor saves its state by pushing its registers and program counter onto the stack. The processor then loads an interrupt *vector* into the program counter. The interrupt vector is the address at which an *Interrupt Service Routine* (*ISR*) lies. Thus, loading the vector into the program counter causes the processor to begin execution of the ISR, performing whatever service the interrupting device required. The last instruction of an ISR is always a *Return from Interrupt* instruction. This causes the processor to reload its saved state (registers and program counter) from the stack and resume its original program. Interrupts are largely transparent to the original program. This means that the original program is completely "unaware" that the processor was interrupted, save for a lost interval of time.

Processors with shadow registers use these to save their current state, rather than pushing their register bank onto the stack. This saves considerable memory accesses (and therefore time) when processing an interrupt. However, since only one set of shadow registers exists, a processor servicing multiple interrupts must "manually" preserve the state of the registers before servicing the higher interrupt. If it does not, important state information will be lost. Upon returning from an ISR, the contents of the shadow registers are swapped back into the main register array.

Hardware interrupts

There are two ways of telling when an I/O device (such as a serial controller or a disk controller) is ready for the next sequence of data to be transferred. The first is *busy waiting* or *polling*, when the processor continuously checks the device's status regis-

ter until the device is ready. This is fairly wasteful of the processor's time but is the simplest to implement.

A better way is for the device to generate an interrupt to the processor when it is ready for a transfer to take place. Small, simple processors may have only one (or two) interrupt input, so several external devices may have to share the interrupt lines of the processor. When an interrupt occurs, the processor must check each device to determine which one generated the interrupt. (This can also be considered a form of polling.) The advantage of interrupt polling over ordinary polling is that the polling occurs only when there is a need to service a device. Polling interrupts is suitable only in systems that have a small number of devices; otherwise, the processor will spend too long trying to determine the source of the interrupt.

The other technique of servicing an interrupt is by using *vectored interrupts*, by which the interrupting device is able to specify which interrupt vector the processor is to execute. Vectored interrupts considerably reduce the time it takes the processor to determine the source of the interrupt. If an interrupt request can be generated from more than one source, it is therefore necessary to assign priorities (levels) to the different interrupts. This can be done in either hardware or software, depending on the particular application. In this scheme, the processor has numerous interrupt lines with each interrupt corresponding to a given interrupt vector. So, for example, when an interrupt of priority 7 occurs (interrupt lines corresponding to 7 are asserted), the processor loads vector 7 into its program counter and starts executing the service routine specific for interrupt 7.

Vectored interrupts can be taken one step further. Some processors and devices support the device actually placing the appropriate vector onto the data bus when they generate an interrupt. This means the system can be even more versatile, so that instead of being limited to one interrupt per peripheral, each device can supply an interrupt vector specific for the event that is causing the interrupt. However, the processor must support this feature, and most do not.

Some processors have a feature known as a *fast hardware interrupt*. With this interrupt, only the program counter is saved. It assumes that the ISR will protect the contents of the registers by manually saving their state as required. Fast interrupts are useful when an I/O device requires a very fast response from a processor and cannot wait for the processor to save all its registers to the stack. A special (and separate) interrupt line is used to generate fast interrupts.

Software interrupts

A software interrupt is an interrupt generated by an instruction. It is the lowest priority interrupt and is generally used by programs to request a service to be performed for it by the system software (operating system or firmware).

So why are software interrupts used? Why isn't the appropriate section of code called directly? For that matter, why use an operating system to perform tasks for us at all?

It gets back to compatibility. Jumping to a subroutine is jumping to a specific address. A future version of the system software may not locate the subroutines at the same addresses as earlier versions. By using a software interrupt, our program does not need to know where the routines lie. It relies on the entry in the vector table to direct it to the correct location.

CISC and RISC

There are two major approaches to processor architecture: *Complex Instruction Set Computer* (*CISC*, pronounced "sisk") processors and *Reduced Instruction Set Computer* (*RISC*) processors. Classic CISC processors are the Intel x86, Motorola 68xxx, and National Semiconductor 32xxx processors and, to a lesser degree, the Intel Pentium. Common RISC architectures are the Motorola/IBM PowerPC, the MIPS architecture, Sun's SPARC, the ARM, the ATMEL AVR, and the Microchip PIC.

CISC processors have a single processing unit, external memory, a relatively small register set, and many hundreds of different instructions. In many ways, they are just smaller versions of the processing units of mainframe computers from the 1960s.

The tendency in processor design throughout the late '70s and early '80s had been toward bigger and more complicated instruction sets. Need to input a string of characters from an I/O port? Well, with CISC (80x86 family), there's a *single instruction* to do it! The diversity of instructions in a CISC processor can easily exceed a thousand opcodes in some processors, such as the Motorola 68000. This had the advantage of making the job of the assembly-language programmer easier—you had to write fewer lines of code to get the job done. Since memory was slow and expensive, it also made sense to make each instruction do more. This reduced the number of instructions needed to perform a given function and thereby reduced memory space and the number of memory accesses required to fetch instructions. As memory got cheaper and faster and compilers became more efficient, the relative advantages of the CISC approach began to diminish. One main disadvantage of CISC is that the processors themselves get increasingly complicated, as a consequence of supporting such a large and diverse instruction set. The control and instruction decode units are complex and slow; the silicon is large and hard to produce; they consume a lot of power and therefore generate a lot of heat. As processors became more advanced, the overheads that CISC imposed on the silicon became oppressive.

A given processor feature when considered alone may increase processor performance but may actually decrease the performance of the total system, if it increases the total complexity of the device. It was found that by streamlining the instruction set to the most commonly used instructions, the processors became simpler and faster. Fewer cycles are required to decode and execute each instruction, and the cycles are shorter. The drawback is that more (simpler) instructions are required to perform a task, but this is more than made up for in the performance boost to the processor. For example, if both cycle time and the number of cycles per instruction

are reduced by a factor of 4 each, while the number of instructions required to perform a task grows by 50%, the execution of the processor is sped up by a factor of 8.

The realization of this led to a rethinking of processor design. The result was the RISC architecture, which has led to the development of very high performance processors. The basic philosophy behind RISC is to move the complexity from the silicon to the language compiler. The hardware is kept as simple and fast as possible.

A given complex instruction can be performed by a sequence of much simpler instructions. For example, many processors have an `xor` (exclusive OR) instruction for bit manipulation, and they also have a `clear` instruction to set a given register to zero. However, a register can also be set to zero by xor-ing it with itself. Thus, the separate clear instruction is no longer required. It can be replaced with the already-present `xor`. Further, many processors are able to clear a memory location directly, by writing zeros to it. That same function can be implemented by clearing a register and then storing that register to the memory location. The instruction to load a register with a literal number can be replaced with clearing a register, followed by an `add` instruction with the literal number as its operand. Thus, six instructions (`xor`, `clear` *`reg`*, `clear` *`memory`*, *`load`* `literal`, `store`, and `add`) can be replaced with just three (`xor`, `store`, and `add`).

So the following CISC assembly pseudocode:

```
clear 0x1000    ; clear memory location 0x1000
load  r1,#5     ; load register 1 with the value 5
```

becomes the following RISC pseudocode:

```
xor   r1,r1     ; clear register 1
store r1,0x1000 ; clear memory location 0x1000
add   r1,#5     ; load register 1 with the value 5
```

The resulting code size is bigger, but the reduced complexity of the instruction decode unit can result in faster overall operation. Dozens of such code optimizations exist to give RISC its simplicity.

RISC processors have a number of distinguishing characteristics. They have large register sets (in some architectures exceeding a thousand), thereby reducing the number of times the processor must access main memory. Often-used variables can be left inside the processor, reducing the number of accesses to (slow) external memory. Compilers of high-level languages (such as C) take advantage of this to optimize processor performance.

By having smaller and simpler instruction decode units, RISC processors have fast instruction execution, but this also reduces the size and power consumption of the processing unit. Generally, RISC instructions will take only one or two cycles to execute (this depends greatly on the particular processor). This is in contrast to instructions for a CISC processor, in which instructions may take many tens of cycles to execute. For example, one instruction (integer multiplication) on an 80486 CISC pro-

cessor takes 42 cycles to complete. The same instruction on a RISC processor may take just one cycle. Instructions on a RISC processor have a simple format. All instructions are generally the same length (which makes instruction decode units simpler).

RISC processors implement what is known as a *load/store* architecture. This means that the only instructions that actually reference memory are `load` and `store`. In contrast, many (most) instructions on a CISC processor may access or manipulate memory. On a RISC processor, all other instructions (aside from `load` and `store`) work on the registers only. This facilitates the attribute of RISC processors that (most of) their instructions complete in a single cycle. As a consequence, RISC processors do not have the range of addressing modes that are found on CISC processors.

RISC processors also often have pipelined instruction execution. This means that while one instruction is being executed, the next instruction in the sequence is being decoded, while the third one is being fetched. At any given moment, several instructions will be in the pipeline and in the process of being executed. Again, this gives improved processor performance. Thus, even though not all instructions may take a single cycle to complete, the processor may issue and retire instructions on each cycle, thereby achieving effective single-cycle execution. Some RISC processors have overlapped instruction execution. `load` operations may allow the execution of subsequent, unrelated instructions to continue before the data requested by the `load` has been returned from memory. This allows these instructions to overlap the `load`, thereby improving processor performance.

Due to their computing power and low power consumption, RISC processors are becoming widely used, particularly in embedded computer systems, and many RISC attributes are appearing in what are traditionally CISC architectures (such as with the Intel Pentium). Ironically, many RISC architectures are adding some CISC-like features, and so the distinction between RISC and CISC is blurring.

An excellent discussion of RISC architectures and processor performance topics can be found in Kevin Dowd and Charles Severance's *High Performance Computing*, available from O'Reilly & Associates.

So, which is better for embedded and industrial applications, RISC or CISC? If power consumption needs to be low, then RISC is probably the better architecture to use. However, if the available space for program storage is small, then a CISC processor may be a better alternative, since CISC instructions get more "bang" for the byte.

Digital Signal Processors

A special type of processor architecture is that of the *Digital Signal Processor* (*DSP*). These processors have instruction sets and architectures optimized for numerical processing of array data. They often extend the Harvard architecture concept further, not only by having separate data and code spaces, but also by splitting the data spaces into two or more banks. This allows concurrent instruction fetch and data

accesses for multiple operands. As such, DSPs can have very high throughput and can outperform both CISC and RISC processors in certain applications.

DSPs have special hardware well suited to numerical processing of arrays. They often have *hardware looping*, whereby special registers allow for and control the repeated execution of an instruction sequence. This is also often known as *zero-overhead looping*, since no conditions need to be explicitly tested by the software as part of the looping process. DSPs often have dedicated hardware for increasing the speed of arithmetic operations. High-speed multipliers, multiply-and-accumulate (MAC) units, and barrel shifters are common features.

DSP processors are commonly used in embedded applications, and many conventional embedded microcontrollers include some DSP functionality.

Memory

Memory is used to hold data and software for the processor. There is a variety of memory types, and often a mix is used within a single system. Some memory will retain its contents while there is no power, yet will be slow to access. Other memory devices will be high capacity, yet will require additional support circuitry and will be slower to access. Still other memory devices will trade capacity for speed, giving relatively small devices, yet are capable of keeping up with the fastest of processors.

Memory can be organized in two ways, either *word-organized* or *bit-organized*. In the word-organized scheme, complete nybbles, bytes, or words are stored within a single component, whereas with bit-organized memory, each bit of a byte or word is allocated to a separate component (Figure 1-10).

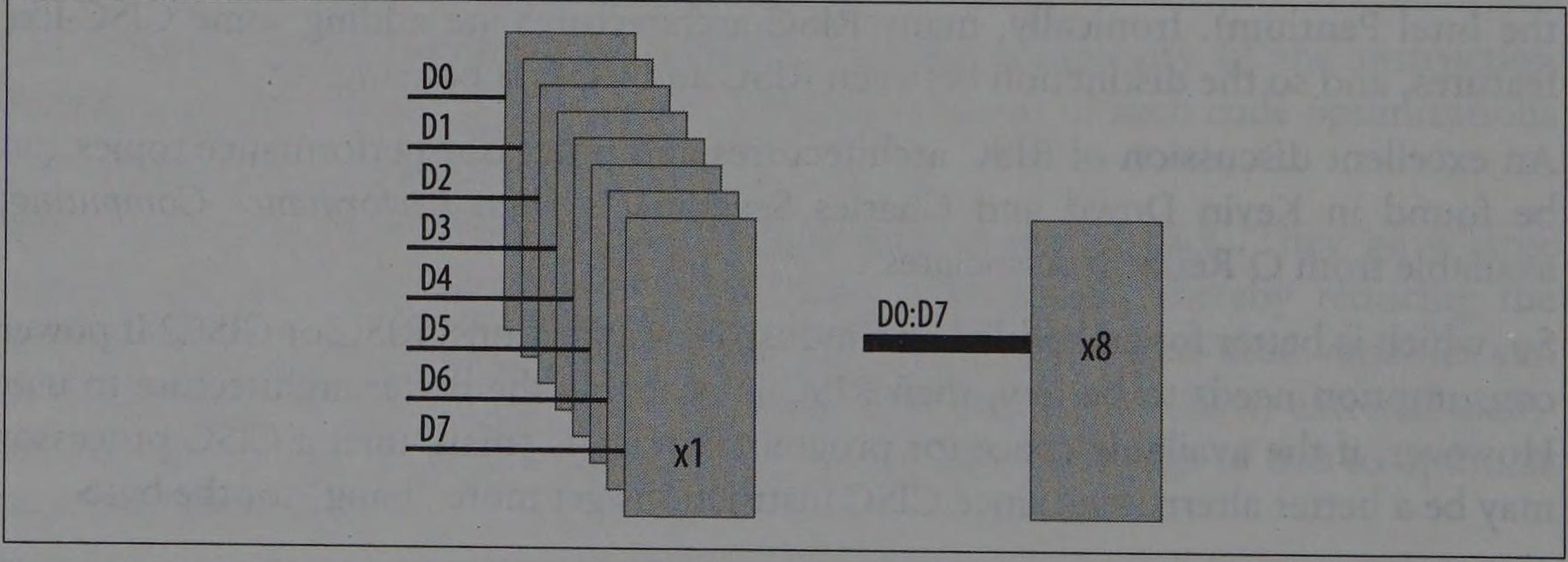


Figure 1-10. Eight bit-organized 8 x 1 devices and one word-organized 1 x 8 device

Memory chips come in different sizes, with the width specified as part of the size description. For instance, a DRAM (dynamic RAM) chip might be described as being 4M x 1 (bit-organized), whereas a SRAM (static RAM) may be 512k x 8 (word-organized). In both cases, each chip has exactly the same storage capacity, but they are

organized in different ways. In the DRAM case, it would take eight chips to complete a memory block for an 8-bit data bus, whereas the SRAM requires only one chip.

However, because the DRAMs are organized in parallel, they are accessed simultaneously. The final size of the DRAM block is (4M x 1) x 8 devices, which is 32M. It is common practice for multiple DRAMs to be placed on a *memory module*. This is the common way that DRAMs are installed in standard computers.

The common widths for memory chips are x1, x4, and x8, although x16 devices are available.

RAM

RAM stands for *Random Access Memory*. This is a bit of a misnomer, since most (all) computer memory may be considered "random access." RAM is the "working memory" in the computer system. It is where the processor may easily write data for temporary storage. RAM is generally *volatile*, losing its contents when the system loses power. Any information stored in RAM that must be retained must be written to some form of permanent storage before the system powers down. There are special nonvolatile RAMs that integrate a battery-backup system, so that the RAM remains powered even when its computer system has shut down.

RAMs generally fall into two categories—*static RAM* (also known as *SRAM*) and *dynamic RAM* (also known as *DRAM*).

Static RAMs use pairs of logic gates to hold each bit of data. Static RAMs are the fastest form of RAM available, require little external support circuitry, and have relatively low power consumption. Their drawbacks are that their capacity is considerably less than dynamic RAM, yet they are much more expensive. Their relatively low capacity requires more chips to be used to implement the same size memory. A modern PC built using nothing but static RAM would be a *considerably* bigger machine and would cost a small fortune to produce. (It would be *very* fast, however.)

Dynamic RAM uses arrays of what are essentially capacitors to hold individual bits of data. The capacitor arrays will hold their charge only for a short period of time before it begins to diminish. Therefore, dynamic RAMs need continuous refreshing, every few milliseconds or so. This perpetual need for refreshing requires additional support and also can delay processor access to the memory. If a processor access conflicts with the need to refresh the array, the refresh cycle must take precedence.

Dynamic RAMs are the highest capacity memory devices available and come in a wide and diverse variety of subspecies. Interfacing DRAMs to small microcontrollers is generally not possible, and certainly not practical. Most processors with large address spaces include support for DRAMs. Connecting DRAMs to such processors is simply a case of connecting the dots (or pins, as the case may be). For processors that do not include DRAM support, special DRAM controller chips are available that make interfacing the DRAMs very simple indeed.

Many processors have instruction and/or data *caches*, which store recent memory accesses. These caches are often internal to the processors and are implemented with fast memory cells and high-speed data paths. Instruction execution normally runs out of the instruction cache, providing for fast execution. The processor is capable of rapidly reloading the caches from main memory should a cache miss occur. Some processors have logic that is able to anticipate a cache miss and begin the cache reload prior to the cache miss occurring. Caches are implemented using fast SRAM and are most often used to compensate for the slowness of the main DRAM array in large systems.

ROM

ROM stands for *Read-Only Memory*. This is also a bit of a misnomer, since many (modern) ROMs can also be written to. ROMs are nonvolatile memory, requiring no current to retain their contents. They are generally slower than RAM and considerably slower than static RAM.

The primary purpose of ROM within a system is to hold the code (and sometimes data) that needs to be present at power-up. Such software is generally known as *firmware*, and contains software to initialize the computer by placing I/O devices into a known state, may contain either a bootloader program to load an operating system off disk or network, or, in the case of an embedded system, may contain the application itself.

Many microcontrollers contain on-chip ROM, thereby reducing component count and simplifying system design.

Standard ROM is fabricated (in a simplistic sense) from a large array of diodes. The unwritten state for a ROM is all 1s, each byte location reading as `0xFF`. The process of loading software into a ROM is known as *burning the ROM*. This term comes from the fact that the programming process is performed by passing a sufficiently large current through the appropriate diodes to "blow them" or *burn* them, thereby creating a zero at that bit location. A device known as a *ROM burner* can accomplish this, or if the system supports it, the ROM may be programmed in-circuit. This is known as *In-System Programming* (*ISP*), or sometimes, *In-Circuit Programming* (*ICP*).

One-Time Programmable (*OTP*) ROMs, as the name implies, can be burned only once. Computer manufacturers typically use them in systems in which the firmware is stable and the product is shipping in bulk to customers. *Mask-programmable* ROMs are also one-time programmable, but unlike OTPs, they are burned by the chip manufacturer prior to shipping. Like OTPs, they are used once the software is known to be stable and have the advantage of lowering production costs for large shipments.

EPROM

OTP ROMs are great for shipping in final products, but they are wasteful for debugging, since, with each iteration of software, a new chip must be burned and the old one thrown away. As such, OTPs make for a very expensive development option.

A better choice for system development and debugging is the *Erasable Programmable Read-Only Memory*, or EPROM. Shining ultraviolet light through a small window on the top of the chip can erase the EPROM, allowing it to be reprogrammed and reused. They are pin and signal compatible with comparable OTP and mask devices. Thus, an EPROM can be used during development, while OTPs can be used in production, with no change to the rest of the system.

EPROMs and their equivalent OTP cousins range in capacity from a few kilobytes (exceedingly rare these days) to a megabyte or more.

The drawback with EPROM technology is that the chip must be removed from the circuit to be erased, and the erasure can take many minutes to complete. Then the chip is placed in the burner, loaded with software, and placed back in circuit. This can lead to very slow debugging cycles. Further, it makes the device useless for storing changeable system parameters.

EEROM

EEROM is *Electrically Erasable Read-Only Memory* and is also known as *EEPROM* (*Electrically Erasable Programmable Read-Only Memory*). Very rarely it is also called *Electrically Alterable Read-Only Memory* (*EAROM*). EEROM can be pronounced as either "e-e ROM" or "e-squared ROM" or sometimes just "e-squared" for short.

EEROMs can be erased and reprogrammed in-circuit. Their capacity is significantly smaller than standard ROM (typically only a few kilobytes), and so they are not suited to holding firmware. They are typically used instead for holding system parameters and mode information, to be retained during power-off.

It is common for many microcontrollers to incorporate a small EEROM on-chip for holding system parameters. This is especially useful in embedded systems and may be used for storing network addresses, configuration settings, serial numbers, servicing records, and so on.

Flash

Flash is the newest ROM technology and is rapidly becoming dominant. Flash memory has the reprogrammability of EEROM and the large capacity of standard ROMs. Flash chips are sometimes referred to as "flash ROMs" or "flash RAMs." Since they are not like standard ROMs nor standard RAMs, I prefer to just call them "flash" and save on the confusion.

Flash is normally organized as sectors and has the advantage that individual sectors may be erased and rewritten without affecting the contents of the rest of the device. Typically, before a sector can be written, it must be erased. It can't just be written over as with a RAM.

There are several different flash technologies, and the erasing and programming requirements of flash devices vary from manufacturer to manufacturer.

Input/Output

The address space of the processor can contain devices other than memory. These are input/output devices (I/O devices, also known as *peripherals*) and are used by the processor to communicate with the external world. Some examples are serial controllers that communicate with keyboards, mice, modems, and so on, and parallel I/O devices that control some external subsystem or disk drive controllers, video and audio controllers, or network interfaces.

There are three main ways in which data may be exchanged with the external world:

Programmed I/O
: The processor accepts or delivers data at times convenient to it (the processor).

Interrupt-driven I/O
: External events control the processor by requesting the current program be suspended and the external event be serviced. An external device will interrupt the processor (assert an interrupt control line into the processor), at which time the processor will suspend the current task (program) and begin executing an interrupt service routine. The service of an interrupt may involve transferring data from input to memory or from memory to output.

Direct Memory Access (DMA)
: DMA allows data to be transferred from I/O devices to memory directly without the continuous involvement of the processor. DMA is used in high-speed systems, in which the rate of data transfer is important. Not all processors support DMA.

DMA

Direct Memory Access is a way of streamlining transfers of large blocks of data between two sections of memory or between memory and an I/O device. Let's say you want to read in 100 MB from disk and store it in memory. You have two options.

The processor can read each byte at a time from the disk controller into a register, then store the contents of the register to the appropriate memory location. For each byte transferred, the processor must read an instruction, decode the instruction, read

the data, read the next instruction, decode the instruction, and then store the data. Then the process starts over again for the next byte.

The second option in moving large amounts of data around the system is DMA. A special device, called a *DMA controller* (*DMAC*), performs high-speed transfers between memory and I/O devices. Using DMA bypasses the processor by setting up a *channel* between the I/O device and the memory. Thus, data is read from the I/O device and written into memory without the need to execute code to perform the transfer on a byte-by-byte (or word-by-word) basis.

In order for a DMA transfer to occur, the DMAC must have use of the address and data buses. There are several ways in which this could be implemented by the system designer. The most common approach (and probably the simplest) is to suspend the operation of the processor and for the processor to release its buses (the buses are tristate). This allows the DMAC to take over the buses for the short period required to perform the transfer. Processors that support DMA usually have a special control input that enables a DMAC (or some other processor) to request the buses.

There are four basic types of DMA:

- *Standard block transfer* is accomplished by the DMA controller performing a sequence of memory transfers. The transfers involve a load operation from a source address followed by a store operation to a destination address. Standard block transfers are initiated under software control and are used for moving data structures from one region of memory to another.
- *Demand-mode transfer* is similar to standard mode except that the transfer is controlled by an external device. Demand-mode transfers are used to move data between memory and I/O or vice versa. The I/O device requests and synchronizes the movement of data.
- *Fly-by transfer* provides high-speed data movement in the system. Instead of using multiple bus accesses as with conventional DMA transfers, fly-by transfers move data from source to destination in a single access. The data is not read into the processor before going to its destination. During a fly-by transfer, memory and I/O are given different bus control signals. For example, an I/O device is given a read request at the same time that memory is given a write request. Data moves from the I/O device straight into the memory device.
- *Data-chaining transfers* allow DMA transfers to be performed as specified by a linked list in memory. Data chaining is started by specifying a pointer to a descriptor in memory. The descriptor is a table specifying byte count, source address, destination address, and a pointer to the next descriptor. The DMAC loads the relevant information about the transfer from this table and begins moving data. The transfer continues until the number of bytes transferred is equal to the entry in the byte count field. On completion, the pointer to the next descriptor is loaded. This continues until a null pointer is found.

To illustrate the use of DMA, let's consider the example of a fly-by transfer of data from a hard disk controller to memory. A DMA transfer begins by the processor configuring the DMAC for the transfer. This setup involves specifying the source, destination, and size of the data, as well as other parameters. The disk controller generates a request for service to the DMAC (not the processor). The DMAC then generates a **HOLD** or **BR** (bus request) to the processor. The processor completes the current instruction; places the address, control, and data buses in a high-impedance state (*floats*, tristates, or releases them); responds to the DMAC with a **HOLD-acknowledge** or **BG** (bus granted); and enters a dormant state. Upon receiving a **HOLD-acknowledge**, the DMAC places the address of the memory location at which the transfer to memory will begin onto the address bus and generates a **WRITE** to the memory, while the disk controller places the data on the data bus. Hence, a direct memory access is accomplished from the disk controller to the memory.

In a similar fashion, transfers from memory to I/O devices are also possible. DMACs are capable of handling block transfers of data. The DMAC automatically increments the address on the address bus to point to each successive memory location as the I/O device generates (or receives) data. Once the transfer is complete, the buses are returned to the processor, and it resumes normal operation.

Not all DMA controllers support all forms of DMA. Some DMA controllers simply read data from a source, hold it internally, and then store it to a destination. They perform the transfer in exactly the same way that a processor would. The advantage of a DMA controller over a processor is that each transfer performed by a processor still has program fetches associated with it. Thus, even though a transfer by a DMA controller takes place by sequential reads and writes, the controller does not also have to fetch and execute code, thereby providing a faster transfer.

Support for DMA is normally not found in small microcontrollers. Some midrange processors (16-bit, low-end 32-bit) may have DMA support. All high-end processors (32-bit and above) will have DMA support, and many include a DMA controller on-chip. Similarly, peripherals intended for small-scale computers will not provide DMA support, whereas peripherals intended for high-speed and powerful computers definitely *will* have DMA support.

Parallel and Distributed Computers

Some embedded applications require greater performance than is achievable from a single processor. For cost reasons, implementing a design with the latest superscalar RISC processor may not be practical, or perhaps the application lends itself to distributed processing with the tasks run across several communicating machines. Using a fleet of lower-cost processors, distributed throughout the installation, may make more sense. Implementing embedded systems using parallel processors is becoming increasingly common.

Introduction to parallel architectures

The traditional architecture for computers follows the conventional, von Neumann serial architecture. Computers based on this form usually have a single, sequential processor. The main limitation of this form of computing architecture is that the conventional processor is able to execute only one instruction at a time. Algorithms that run on these machines must therefore be expressed as a sequential problem. A given task must be broken down into a series of sequential steps, each to be executed in order, one at a time.

Many problems that are computationally intensive are also highly parallel. An algorithm that is applied to a large data set characterizes these problems. Often the computation for each element in the data set is the same and is only loosely reliant on the results from computations on neighboring data. Thus, speed advantages may be gained from performing calculations in parallel for each element in the data set, rather than sequentially moving through the data set and computing each result in a serial manner. Machines with multitudes of processors working on a data structure in parallel often far outperform conventional computers in such applications.

The *grain* of the computer is defined as the number of processing elements within the machine. A *coarsely grained* machine has relatively few processors, whereas a *finely grained* machine may have tens of thousands of processing elements. Typically, the processing elements of a finely grained machine are much less powerful than those of a coarsely grained computer. The processing power is achieved through the brute-force approach of having such a large number of processing elements.

There are several different forms of parallel machine. Each architecture has its own advantages and limitations, and each has its share of supporters.

Single-instruction multiple-data computers

Single-Instruction Multiple-Data (*SIMD*) computers are highly parallel machines, employing large arrays of simple processing elements. In an SIMD machine, each processing element has a small amount of local memory. The instructions executed by the SIMD computer are broadcast from a central instruction server to every processing element within the machine. In this way, each processor executes the same instruction as all other processing elements within the machine. Since each processor executes the instruction on its local data, all elements within the data structure are worked upon simultaneously.

The SIMD machine is generally used in conjunction with a conventional computer. An example of this was the Connection Machine (CM-1) by Thinking Machines Corporation, which used either a VAX minicomputer or a Silicon Graphics or Sun workstation as the "host" computer. The Connection Machine was a finely grained SIMD computer with up to 64K processing elements that appeared as a block of 64K of "intelligent memory" to the host system. An application running on the host downloaded a data set into the processor array of the Connection Machine, each processor

within the CM-1 acting as a single memory unit. The host then issued instructions to each processing element of the CM-1 simultaneously. After the computations were completed, the host then read back the result from the Connection Machine, as though it were conventional memory.

The primary advantage of the SIMD machine is that simple and cheap processing elements are used to form the computer. Thus, significant computing power is available using inexpensive, off-the-shelf components. In addition, since each processor is executing the same instructions and therefore sharing a common instruction fetch, the architecture of the machine is somewhat simpler. Only one instruction store is required for the entire computer.

The use of multiple processing elements, each executing the same instructions in unison, is also the SIMD's main disadvantage. Many problems do not lend themselves to being broken down into a form suitable for executing on an SIMD computer. In addition, the data sets associated with a given problem may not match well with a given SIMD architecture. For example, an SIMD machine with 10k processing elements does not mesh well with a data set of 12k data elements.

Multiple-instruction multiple-data computers

The other major form of parallel machine is the *Multiple-Instruction Multiple-Data* (*MIMD*) computer. These machines are typically coarsely grained collections of semiautonomous processors, each with its own local memory and local programs. An algorithm being executed on an MIMD computer is typically broken up into a series of smaller subproblems, each executed on a processor of the MIMD machine. By giving each processing element in the MIMD machine identical programs to execute, the MIMD machine may be treated as an SIMD computer. The grain of an MIMD computer is much less than that of an SIMD machine. MIMD computers tend to use a smaller number of very powerful processors, rather than a large number of less powerful ones.

MIMD computers can be of one of two types, *shared-memory MIMD* and *message-passing MIMD*. Shared-memory MIMD systems have an array of high-speed processors, each with local memory or cache, and each with access to a large, global memory (Figure 1-11). The global memory contains the programs and data to be executed by the machine. Also in this memory is a table of processes (or subprograms) awaiting execution. Each processor will fetch a process and associated data into its local memory or cache and will run semiautonomously of the other processors in the system. Process communication also takes place through the global memory.

A speed advantage is gained by sharing the program among several powerful processors. However, logic within the system must arbitrate between processors for access to the shared memory and associated shared buses of the system. In addition, allowances must be made for a processor attempting to access data in global memory that is out of date. If processor A reads a process and data structure into its local memory and subse-

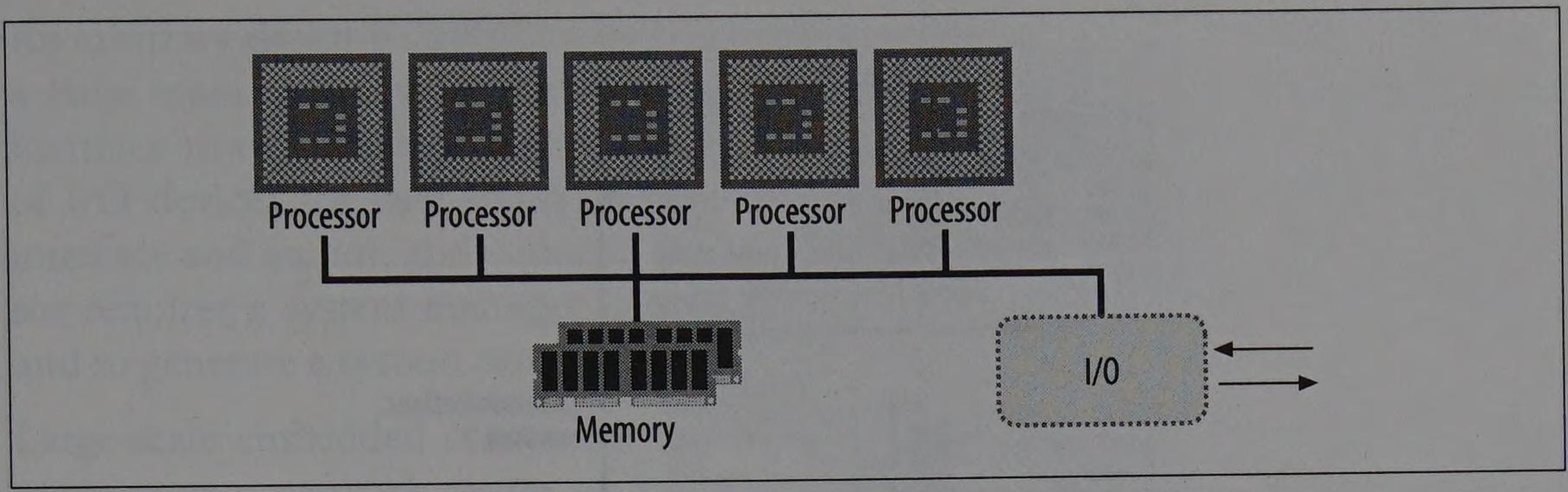


Figure 1-11. Shared-memory MIMD

quently modifies that data structure, processor B attempting to access the same data structure in main memory must be notified that a more recent version of the data structure exists. Such arbitration is implemented in processors like the (now-extinct) Motorola MC88110, which was intended for use in shared-memory MIMD machines.

An alternative MIMD architecture is that of the message-passing MIMD computer (Figure 1-12). In this system, each processor has its own local, main memory. No global memory exists for the machine. Each processing element (processor with local memory) either loads or has loaded into it the programs (and associated data) that it is to execute. Each process runs autonomously on its local processor, and interprocess communication is achieved through message passing through a common medium. The processors may communicate through a single, shared bus (such as Ethernet, CAN, or SCSI) or by using a more elaborate interprocessor connection architecture, such as 2-D arrays, N-dimensional hypercubes, rings, stars, trees, or fully interconnected systems.

Such machines do not suffer the bus contention problems of shared-memory machines. However, the most effective and efficient means of interconnecting the processing nodes of a message-passing MIMD machine is still a major area of research. Each different architecture has its own merits, and which is best for a given application depends to a certain degree on what that application is. Problems that require only a limited amount of interprocess communication may work effectively on a machine without high interconnectivity, whereas other applications may weigh down the communications medium with their message passing. If a percentage of a processing node's time is spent in message routing for its neighbors, a machine with a high degree of interprocess communication but with a low degree of interconnectivity may spend most of its time dealing in message passing with little time spent on actual computation.

The ideal interconnection architecture is that of the fully interconnected system, with every processing node having a direct communications link with every other processing node. However, this is not always practical due to the costs and logistics of such a high degree of interconnectivity. A solution to this problem is to provide each processing element in the machine with a limited number of connections, based on the assumption that a processing element will not need or be able to communicate with

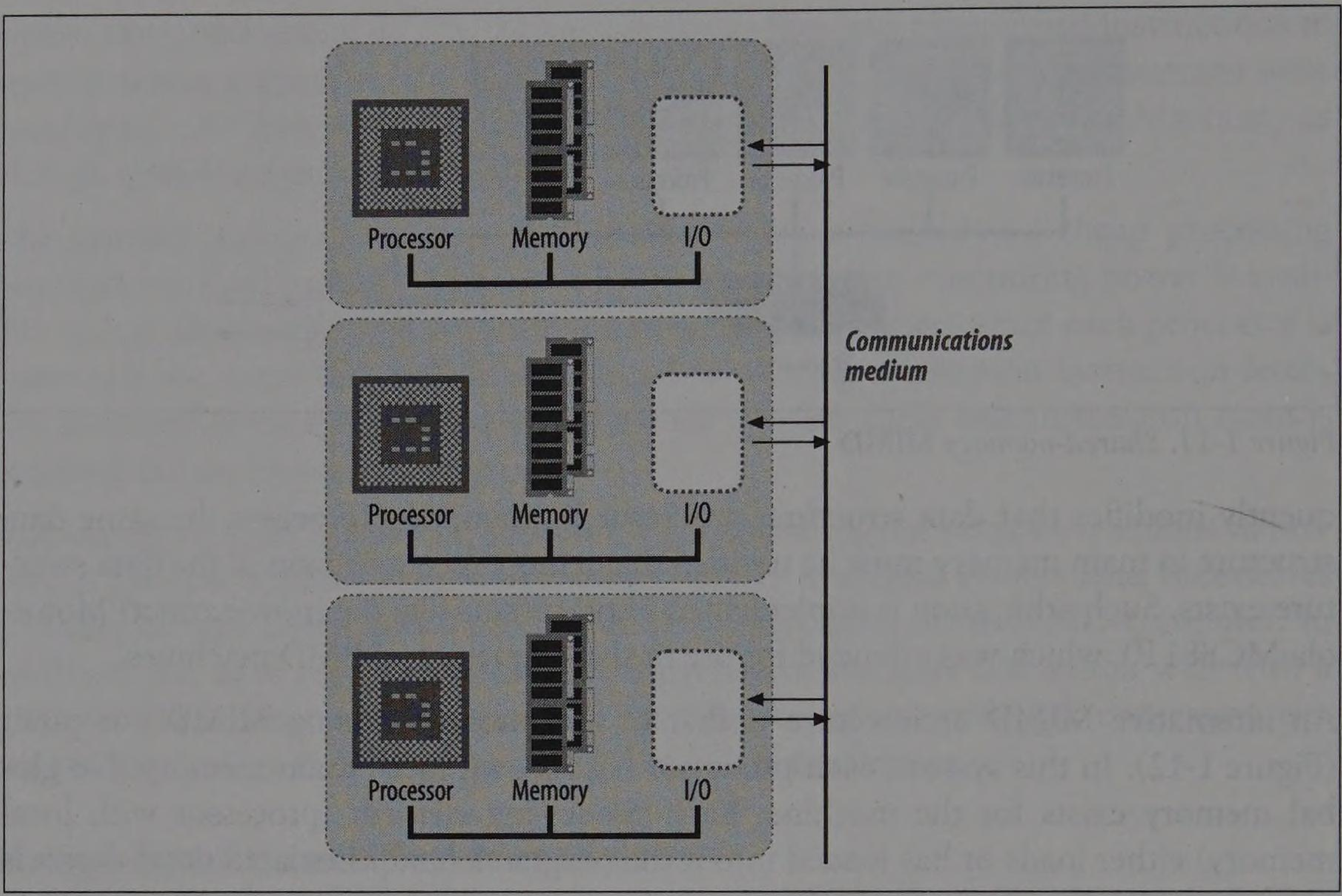


Figure 1-12. Message-passing MIMD

every other processing element in the machine simultaneously. These limited connections from each processing node may then be interconnected using a crossbar switch, thereby providing full interconnectivity for the machine through only a limited number of links per node.

A *distributed* machine is composed of individual computers, networked together as a loosely coupled MIMD parallel machine. Projects such as *Beowulf* and even *SETI@Home* can be considered MIMD machines. Distributed machines are common in the embedded world. A collection of small processing nodes may be distributed across a factory, providing local monitoring and control, and together forming a parallel machine executing the global control algorithm. The avionics of commercial and military aircraft are also distributed parallel computers.

Now let's take a look at computer applications and how that relates to the architecture of the machine.

Embedded Computer Architecture

What a computer is used for, what tasks it must perform, and how it interacts with humans and other systems determine the functionality of the machine, and therefore its architecture, memory, and I/O.

An arbitrary desktop computer (not necessarily a PC) is shown in Figure 1-13. It has a large main memory to hold the operating system, applications, and data and an interface to mass storage devices (disks and DVD/CD-ROMs). It will have a variety of I/O devices for user input (keyboard, mouse, and audio), user output (display interface and audio), and connectivity (networking and peripherals). The fast processor requires a system manager to monitor its core temperature and supply voltages and to generate a system reset.

Large-scale embedded computers may also take the same form. For example, they may act as a network router or gateway and so will require one or more network interfaces, large memory, and fast operation. They may also require some form of user interface as part of their embedded application and, in many ways, may simply be a conventional computer dedicated to a specific task. Thus, in terms of hardware, many high-performance embedded systems are not that much different from a conventional desktop machine.

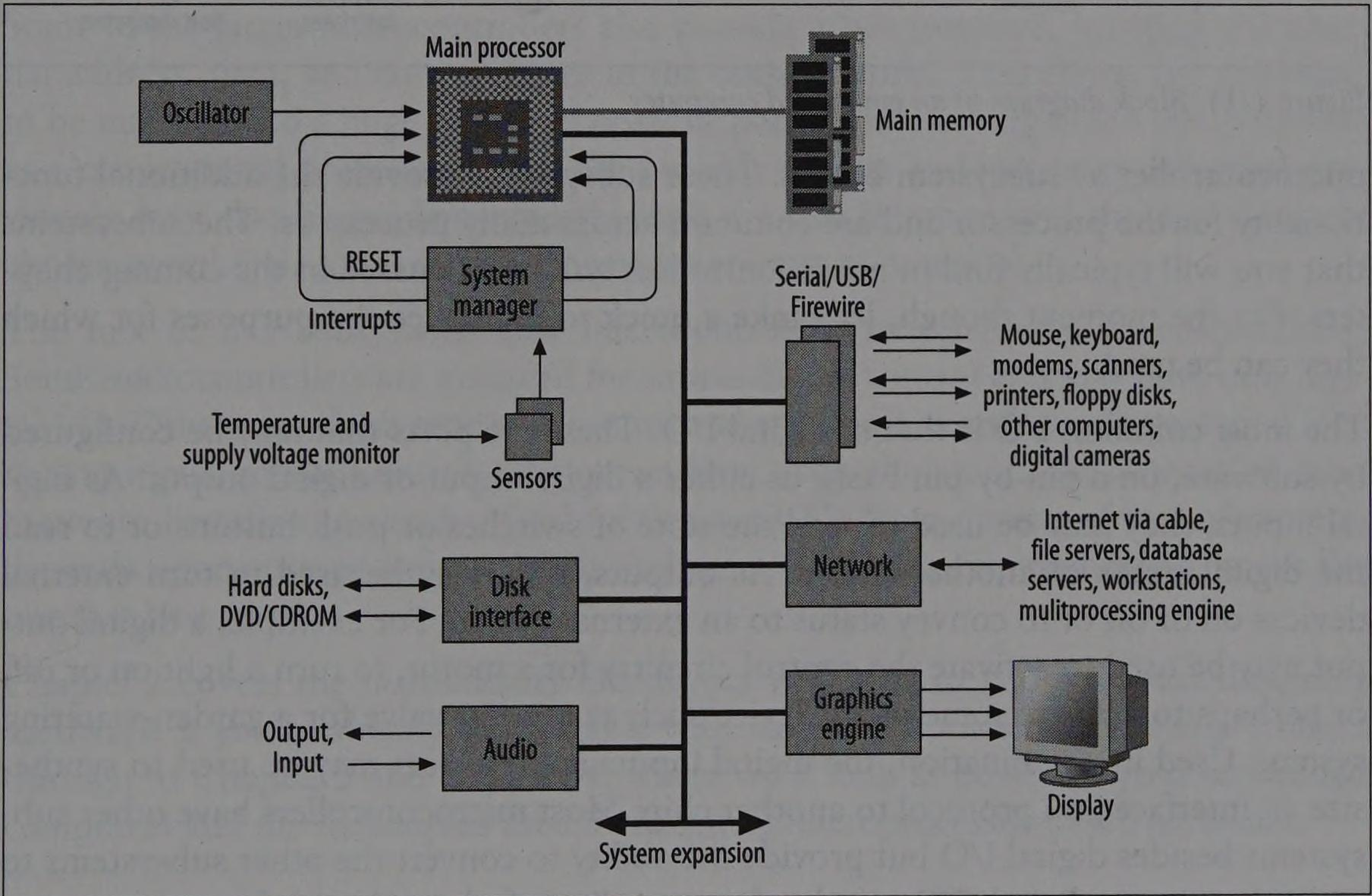


Figure 1-13. Block diagram of a generic computer

Smaller embedded systems use microcontrollers as their processor, with the advantage that this processor will incorporate much of the computer's functionality on a single chip. An arbitrary embedded system, based on a generic microcontroller, is shown in Figure 1-14.

The microcontroller has, at a minimum, a CPU, a small amount of internal memory (ROM and/or RAM), and some form of I/O, which is implemented within a

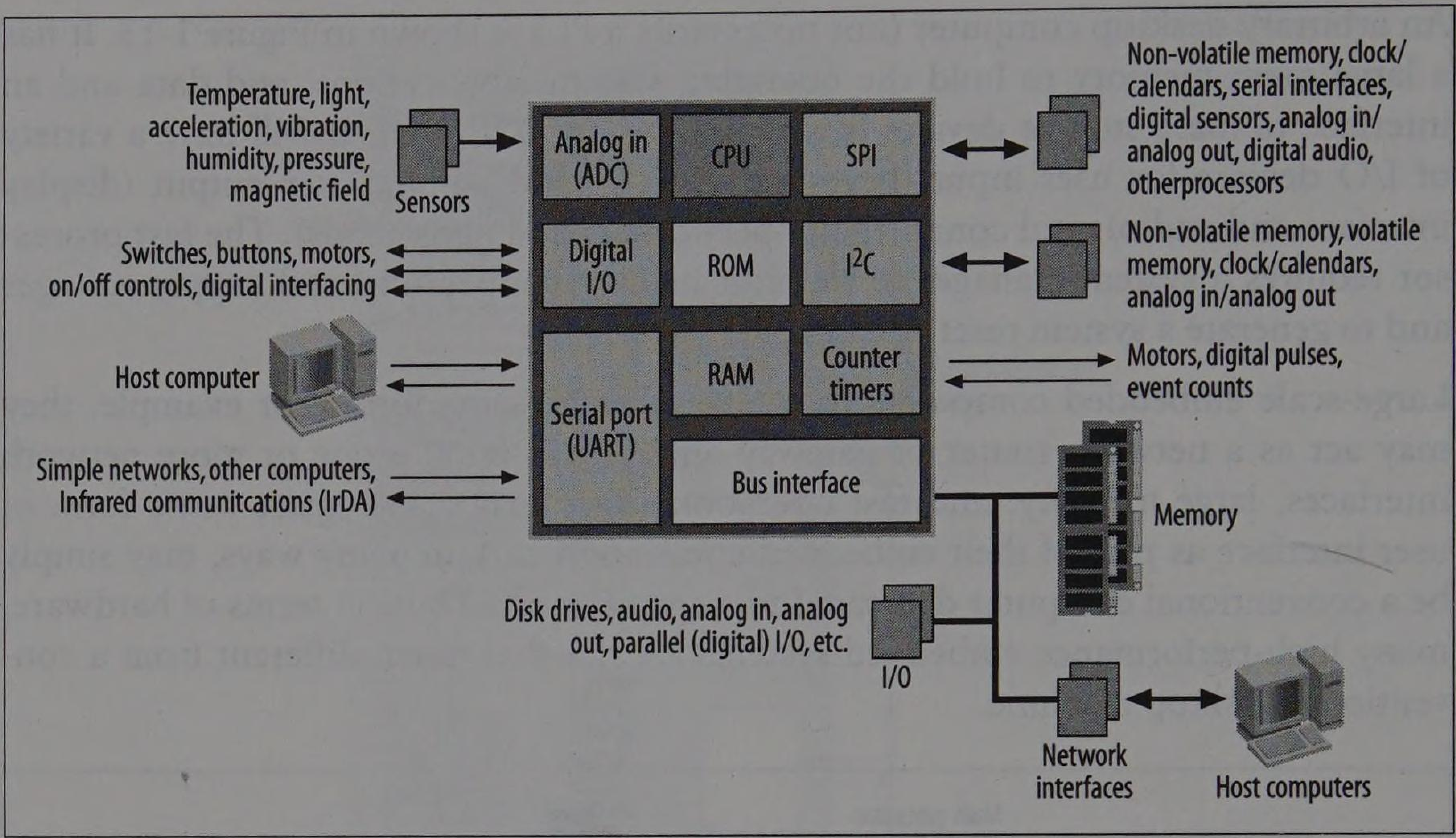


Figure 1-14. Block diagram of an embedded computer

microcontroller as subsystem blocks. These subsystems provide the additional functionality for the processor and are common across many processors. The subsystems that you will typically find in microcontrollers will be discussed in the coming chapters. For the moment though, let's take a quick tour and see the purposes for which they can be used.

The most common I/O is that of digital I/O. These are ports that may be configured by software, on a pin-by-pin basis, as either a digital input or digital output. As digital inputs, they may be used to read the state of switches or push buttons or to read the digital status of another device. As outputs, they may be used to turn external devices on or off or to convey status to an external device. For example, a digital output may be used to activate the control circuitry for a motor, to turn a light on or off, or perhaps to activate some other device such as a water valve for a garden-watering system. Used in combination, the digital inputs and outputs may be used to synthesize an interface and protocol to another chip. Most microcontrollers have other subsystems besides digital I/O but provide the ability to convert the other subsystems to general-purpose digital I/O, if the functionality of the other subsystems is not required. As a system designer, this gives you great versatility in how you use your microcontroller within your application.

Many microcontrollers also have analog inputs, allowing sensors to be sampled for monitoring or recording purposes. Thus, an embedded computer may measure light levels, temperature, vibration or acceleration, air or water pressure, humidity, or magnetic field, to name just some. Alternatively, the analog inputs may be used to monitor simple voltages, perhaps to ensure the reliable operation of a larger system.

Some microcontrollers have serial ports (covered in Chapter 10), which enable the embedded computer to be interfaced to a host computer, another embedded system, or perhaps a simple network. Specialized forms of serial interface, such as SPI and I^2C (Chapter 9), provide a simple way of expanding the microcontroller's functionality. They allow peripherals to be interfaced to the microcontroller, providing access to such devices as off-chip memories (for data or parameter storage), clock/calendar chips (for timekeeping), sensors with digital interfaces, external analog input or output, and even audio chips and other processors.

Most microcontrollers have timers and counters. These may be used to generate internal interrupts at regular intervals, for multitasking, to generate external triggers for off-chip systems, or to provide control pulses for motors. Alternatively, they may be used to count external triggers (pulses) from another system.

A few microcontrollers also include network interfaces such as USB (Chapter 10), Ethernet (Chapter 11), or CAN (Chapter 11).

Some of the larger microcontrollers also provide a bus interface, bringing the internal address, data, and control buses to the outside world. This allows the processor to be interfaced to a huge variety of possible peripherals, in very much the same way as a conventional processor. All of the possible devices and interfaces described previously may also be implemented through the bus interface and the appropriately chosen peripheral. A bus interface provides enormous potential.

The mix of I/O subsystems that microcontrollers may have varies considerably. Some microcontrollers are intended for simple digital control and may have only digital I/O. Others may be intended for industrial applications and may have digital I/O, analog input, motor control, and networking. The choice of microcontroller (and there are literally thousands of subspecies available from dozens of manufacturers) depends upon your processing needs and your interfacing requirements. Choose the one that best suits.

Chapter 2 covers the introductory electronics you need to know to start designing hardware. If you're already comfortable with basic electronics, you can skip straight through to Chapter 3 and Chapter 4, where we'll look at powering your embedded computers and the techniques used in designing microprocessor-based hardware.

CHAPTER 2
Electronics 101

. . . in reality, nothing but atoms and void
—Democritus

In writing this book, my hope is to bring to you an understanding of the design process involved in producing an embedded computer system. To this end, I have kept the electronics, the chips, and the systems I have used as simple as possible. I want you to understand the big picture without getting lost in the details. But, no matter how simple I keep the computer designs, you won't get very far without at least a very rudimentary understanding of electronics. So this chapter presents basic background theory to guide you on your way. Electronics is a truly vast and complex multidisciplinary field, and it is not possible to cover even a thousandth of it in a single chapter. What I will do is to give you an easy-to-understand grounding in the basic principles necessary for embedded computer engineering. The rest of the vast mountain I will leave unvisited. If you want to learn more, pick up a copy of Paul Horowitz and Winfield Hill's *The Art of Electronics*, published by Cambridge University Press. It's a great introductory text. For some fun, interactive online tutorials go to *http://www.clarkson.edu/~svoboda/eta*.

Voltage and Current

It's all about electrons—hence the term, "electronics." Electrons are subatomic particles with a negative charge. They are bound to positively charged atomic nuclei through Coulombic attraction. The classical physics view was to think of electrons "orbiting" the nucleus, analogous to planets orbiting a solar system. While not at all correct,* this makes it easier to visualize what goes on. The strength by which electrons are bound to the nucleus varies from atomic element to atomic element and

* The truth, as always, is far stranger. The quantum view is both beautiful and bizarre. For a simple and elegant introduction, read Richard Feynman's brilliant "QED" (Quantum Electro Dynamics).

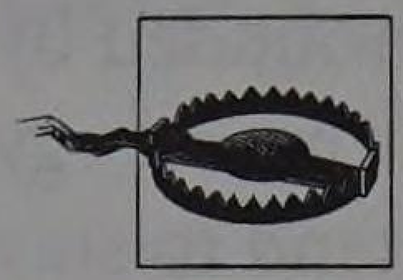
from molecule to molecule. Substances are either *conductors*, *insulators*, or *semiconductors*. In a conductor, such as a metal, the energy required to shift an electron from one nucleus to another is negligible, and the electrons may easily exchange with nearby atomic nuclei. In effect, the metal is a collection of nuclei surrounded by a "sea" of semifree electrons. In an insulator, the opposite is true. The energy required to shift an electron from a nucleus is excessive, and so electrons tend to stay put. In a semiconductor, the substance may act either as a conductor or as an insulator, depending upon external influences. By controlling the external influences, you change the conductivity of the substance, and therefore change the way electrons move within that substance. In effect, a semiconductor is a switch, a switch that may be controlled by other semiconductors. This basic principle is the basis of all modern electronics, the cornerstone upon which everything digital is founded.

The flow of electrons through a conductor or a semiconductor is known as *current*. Current is measured in *Amperes*, more commonly called just plain *Amps* (with the unit symbol A). For an electron to move through a conductor,* there must be a "vacancy" at the next nucleus into which it can shift. (If the next nucleus has a full complement of electrons, the Coulombic repulsion of those electrons will prevent any others from slotting in.) Semiconductor physicists term these vacancies as *holes*. An electron shifting into a neighboring hole leaves a new hole behind it. This new hole is then filled by another electron further down the line, which, in turn, creates another new hole. So current flow is, in effect, a movement of electrons in one direction and a "movement of holes" in another. The electrons are negatively charged, and the holes may be thought of as positive charges. (A missing electron at a nucleus means that the positive charge of the nucleus isn't fully cancelled, and so a net positive charge exists at that location.) So while electrons move from negative to positive, the holes move from positive to negative, and it is the movement of holes (rather than electrons) that we refer to when we talk about current. Current flow, as we work with it in electronics, is deemed to be from positive to negative. For continued current flow, there must be a continuous circular flow of electrons in one direction and holes in the other direction. It is from this circular flow that we derive the term *circuit*.

For current flow to occur between two points, an imbalance must exist between electrons at one end and holes at the other. The size of this imbalance is known as the *potential difference* or *voltage difference* between two points. (It is also sometimes termed "the *voltage drop* across an electronic component.") The unit of voltage difference is the *Volt* (unit symbol V). The greater the voltage difference, the greater the opportunity for current flow. It is very important to note that voltage refers to the *difference* between two points. A voltage cannot exist in isolation. Although you will sometimes see a statement like "the voltage at this point is . . . ," it is a given that the

* I'm treating a conducting semiconductor as though it were an ordinary conductor.

voltage is relative to some common reference point, usually ground (the zero volt reference point).



A common beginner's mistake in testing electronic circuits is to wire up only one lead of a piece of test equipment. Without both leads, there is no common reference point; therefore, any measurement taken is meaningless.

Analog Signals

An analog signal can have an amplitude of any voltage within a range, unlike a digital signal, which can be in one of two defined voltage states (either *high* or *low*). Figure 2-1 shows a typical analog signal (in this case, a sine wave).

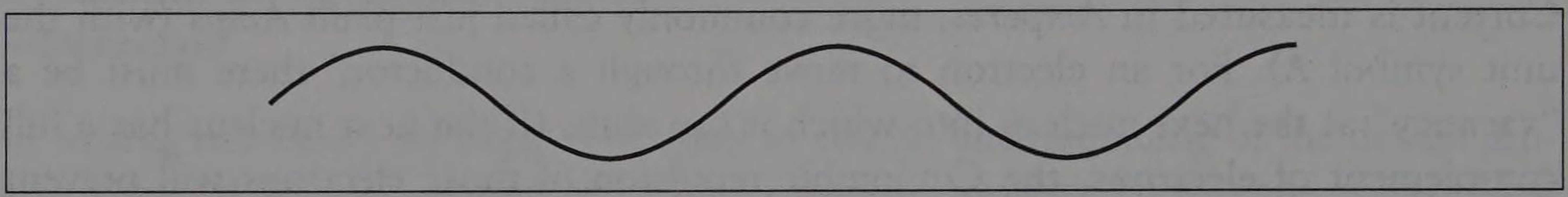


Figure 2-1. An analog waveform

The voltage of a signal may vary over time, or it may be constant. If the voltage varies, it may repeat at regular intervals, in which case the signal is said to be *periodic*. The *period* is the interval of time that it takes the signal pattern to repeat (for example, from one wave crest to another). The *frequency* of the signal is the number of times per second that the pattern repeats.

Frequency is measured in Hertz (Hz) and relates to the period in the following way:

```
Frequency = 1 / Period
```

Thus, a signal with a period of 1ms has a frequency of 1kHz.

A *unipolar signal* (Figure 2-2) has component voltages that are either all positive or all negative. A *bipolar signal* (Figure 2-3) has both positive and negative voltages.

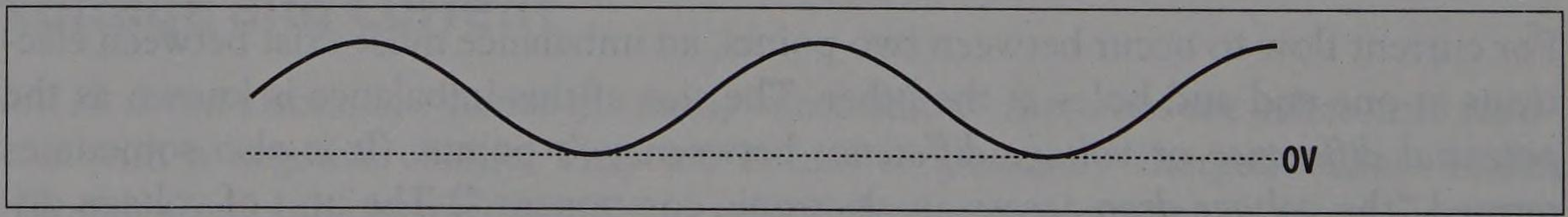


Figure 2-2. Unipolar signal

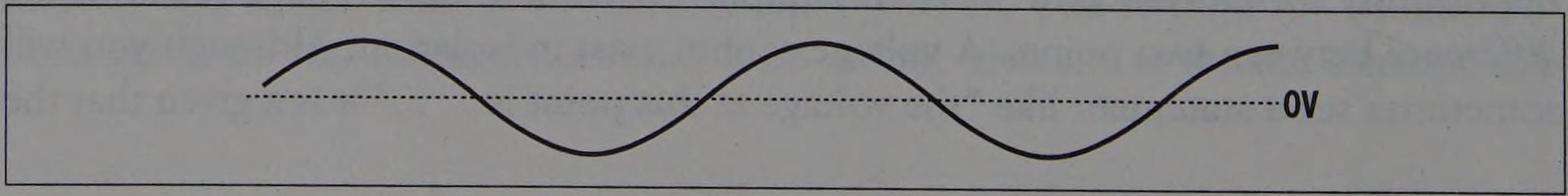


Figure 2-3. Bipolar signal

A typical analog signal will have both an *AC component* and a *DC component* (Figure 2-4). The DC component is the fixed voltage of the signal. The AC component is a varying voltage imposed upon the DC component. The AC component is sometimes referred to as the *peak-to-peak* amplitude of a signal and is denoted with the suffix *pp*. For example, an AC component of 5V would be written as 5Vpp.

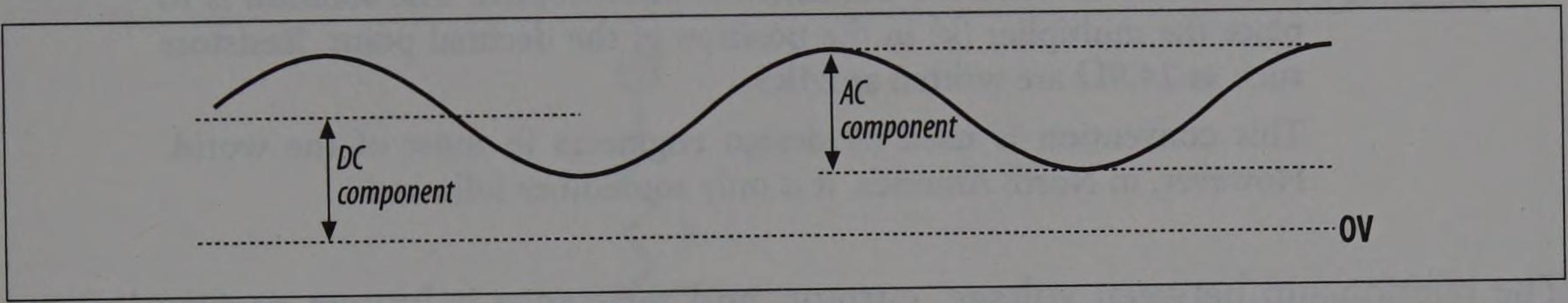


Figure 2-4. DC and AC components of an analog signal

Power

A voltage difference is generated by a difference in potential energy between two points. Therefore, to generate a voltage you use a device that can create such an energy difference. Such devices may be mechanical (generators), which convert motion into a potential difference by electromagnetics, photovoltaic (solar cells), or chemical (batteries). Conversely, a voltage difference (and thereby current flow) can be used to produce mechanical movement (motors), light emission (lightbulbs, LEDs), and heat (toasters, Pentium 4 processors).

Power is the amount of work per time (Joules per second) and is measured in *Watts* (unit symbol W). The equation for calculating power is simply:

```
P = V * I
```

No electronic device is 100% efficient (far from it!), and so it will consume power as it performs its task. The power consumed by a device may be calculated using the preceding equation, from the voltage difference across the device and the current flowing through the device. A typical embedded computer may consume a few hundred mW (milliWatts) of power, but it can vary quite considerably. A large and powerful embedded machine may use several tens (or even hundreds) of Watts, while a tiny embedded controller may use just microWatts.

Resistors

Even a conductor (such as a metal wire) is not 100% efficient at conducting current flow. As current flows through the wire, energy will be lost as heat (and sometimes light). For very small currents, this energy loss is negligible, but for large currents, the loss can cause the conductor to become quite hot (an effect utilized in toasters) or glow brightly (lightbulbs). This loss of energy results in a voltage difference across the wire (or component). The component is said to resist the current flow. This *resistance* (also known as *impedance*, although impedance is somewhat more complex than simple

resistance) is measured in *Ohms* (unit symbol Ω, equation symbol R). Schematics commonly leave off the Ω symbol, so 100kΩ is usually written as just 100k.

> On a schematic, a 4.7kΩ value may be written not as 4.7k, but rather as 4k7. The reason is that it is too easy for a decimal point to be missed or lost when the document is photocopied. The solution is to place the multiplier (k) in the position of the decimal point. Resistors such as 24.9Ω are written as 24R9.
>
> This convention is used by design engineers in most of the world. However, in North America, it is only sometimes followed.

The relationship between voltage, current, and resistance is known as *Ohm's Law*, and is given by:

```
V = I * R
```

For a fixed resistance, a varying voltage will produce a varying current, while a constant voltage will produce a constant current. Hence, a varying voltage source is known as an *Alternating Current* source (or *AC*), while a constant voltage source is known as a *Direct Current* source (*DC*). An AC voltage is normally specified as *VAC*, while a DC voltage is either *VDC* or more often just *V*.

> The stuff that comes out of your wall socket is AC and is nominally 110–120VAC (at 60Hz) if you live in North America, 100VAC if you're in Japan (50Hz in the eastern half—Tokyo—and 60Hz in the western half—Osaka, Kyoto, and Nagoya), and 220–240VAC (at 50Hz) if you're in Australia, New Zealand, the UK, or Europe. All digital electronics, and that includes computers, use DC internally and operate at typical voltages of either 5V or 3.3V. (Some digital electronics will operate at voltages as low as 1.8V or even lower.) The *power supply* of the computer (or TV or stereo or . . .) converts the high-voltage AC supply into the lower DC required by the electronics. The AC adaptor or plug pack (charger) for your cell phone is also an example of a power supply.

For a given voltage difference, the smaller the resistance, the larger the current flow. Conversely, the bigger the resistance, the smaller the current flow. In this way, resistance can be used to limit the current flow through a particular part of a circuit. Special components, known as *resistors*, are produced for precisely this purpose. The schematic component symbols for a resistor are shown in Figure 2-5. Both symbols mean the same thing. The more commonly seen symbol is on the left.

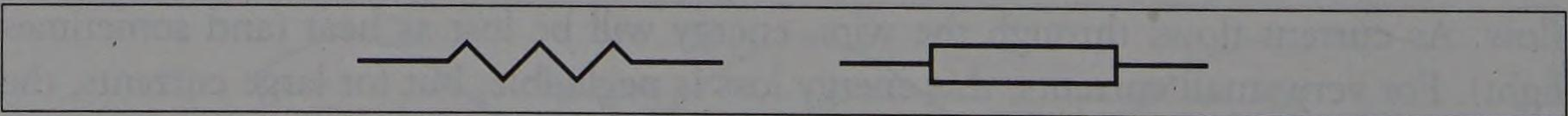


Figure 2-5. Resistor symbols

A resistor may be used to *pull up* (or *pull down*) a signal line to a given voltage level. Figure 2-6 shows a pull-up resistor and a push button. When the button is open (not

pressed), there is no current flow through the resistor, therefore the voltage at V_{OUT} is (in this case) +5V. (Since there is no current flow through the resistor, there is no voltage drop across it.) When the button is pushed, V_{OUT} is connected to ground, and as a consequence, current will flow through the resistor. This simple circuit can be used to switch an input between two logic-level thresholds.

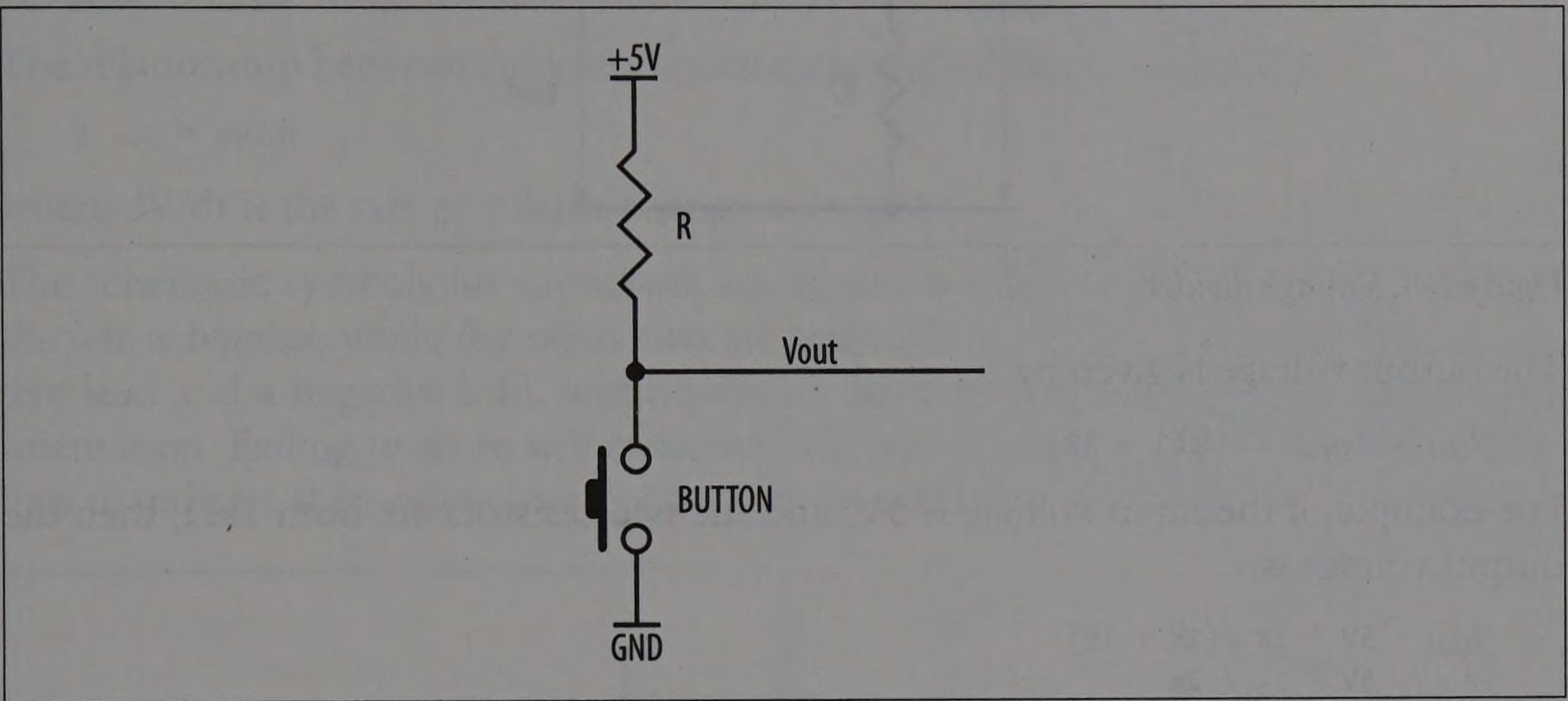


Figure 2-6. Pull-up resistor and a push button

Resistors may be combined together to increase resistance. This is known as a *series* connection (Figure 2-7).

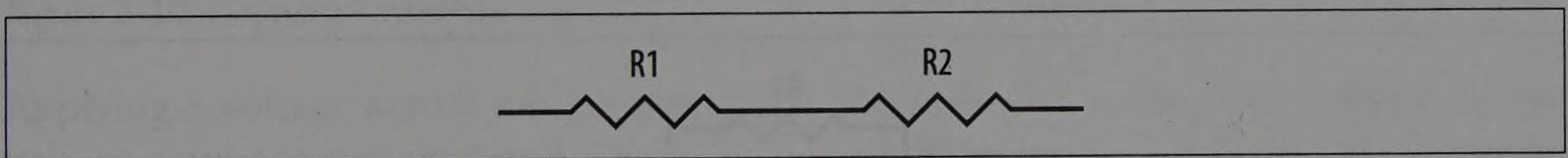


Figure 2-7. Resistors in series

The combined total resistance is given by the relation:

$$R_{TOTAL} = R1 + R2$$

The current flow through *any* of the components in series connection will be the same for each component. In other words, the current flowing through the first resistor will be the same as through the second resistor. This derives from *Kirchhoff's Current Law*.

> **Kirchhoff's Current Law**
>
> The current flowing through a given circuit point is equal to the sum of the currents flowing into that circuit point and is also equal to the sum of currents flowing out of that circuit point.
>
> In other words, what flows in must flow out.

Resistors may be used in a *voltage divider* (Figure 2-8), to provide an intermediate voltage.

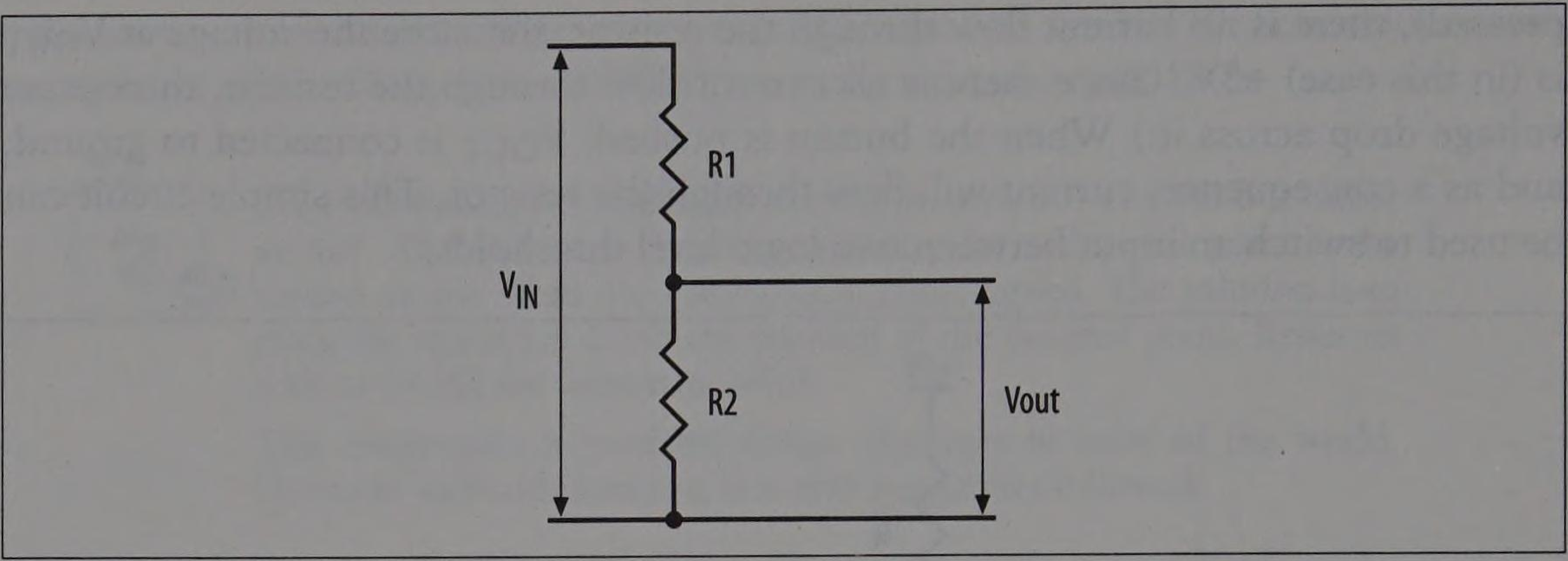


Figure 2-8. Voltage divider

The output voltage is given by:

```
VOUT = VIN * R2 /(R1 + R2)
```

For example, if the input voltage is 5V, and the two resistors are both 1kΩ, then the output voltage is:

```
VOUT = 5V * 1k /(1k + 1k)
     = 5V * 1k / 2k
     = 5V * 0.5
     = 2.5V
```

As you would expect, a voltage divider using equal resistors halves the input voltage.

Resistors combined in *parallel* (Figure 2-9) will decrease the total resistance.

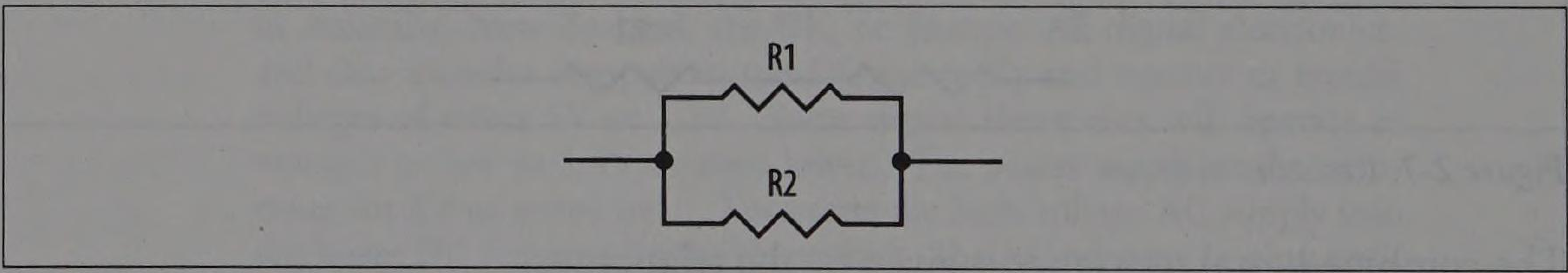


Figure 2-9. Resistors in parallel

The combined total resistance is given by the relation:

```
RTOTAL = 1 / (1/R1 + 1/R2)
```

The voltage drop across R1 must be the same as the voltage drop across R2. However, unless R1 is equal to R2 (and there is no requirement for them to be equal), the current flows through each will be different. This is derived from *Kirchhoff's Voltage Law*.

Kirchhoff's Voltage Law

The sum of the voltage differences around a closed circuit is zero.

Resistors are part of a family of devices known as *passive components*. The other common passive component is the capacitor.

Capacitors

While a resistor is a component that resists the flow of charge through it, a capacitor stores charge. Capacitance is measured in *Farads* (or more formally, *Faradays*) with an equation symbol C and a unit symbol F. Typical capacitors you will use will range in value from µF (microFarads) down to pF (picoFarads).

The relationship between current, capacitance, and voltage is given by:

```
I = C * dV/dt
```

where dV/dt is the rate of voltage change over time.

The schematic symbols for capacitors are shown in Figure 2-10. The component on the left is *bipolar*, while the other two are *unipolar*. A unipolar capacitor has a positive lead and a negative lead, and it must be inserted into a circuit with the correct orientation. Failing to do so will cause it to explode. (Unipolar capacitors have markings to indicate their orientation.) A bipolar capacitor has no polarity.

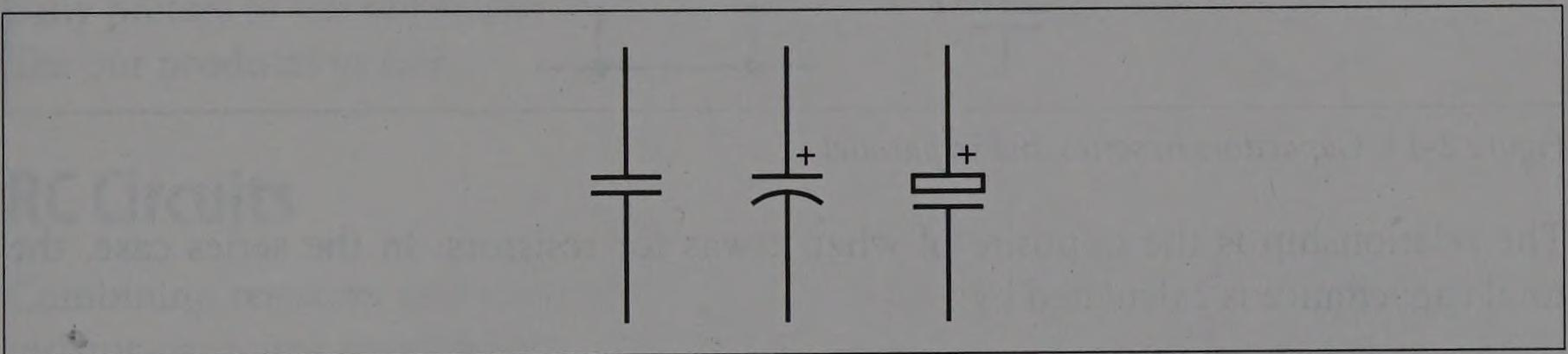


Figure 2-10. Capacitor symbols

Applying a voltage across a capacitor causes the capacitor to become charged. If the voltage source is removed and a path for current flow exists elsewhere in the circuit, the capacitor will discharge and thereby provide a (temporary) voltage and current source (Figure 2-11).

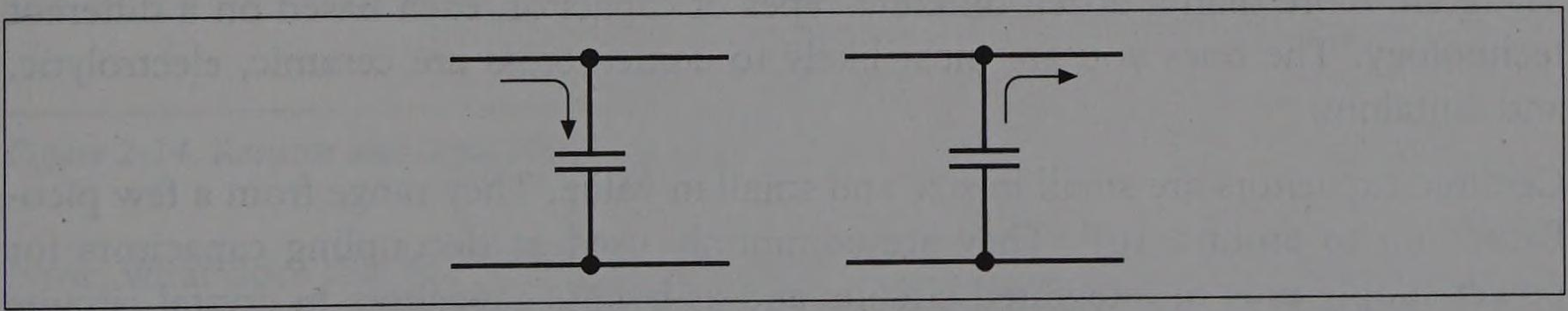


Figure 2-11. Capacitor charging and discharging

This is an extremely useful characteristic. A given voltage source may have a *DC component* (a fixed voltage) and an *AC component* (a ripple voltage superimposed). (Here *component* does not mean a physical device, but rather a fractional part of a voltage.) The capacitor becomes charged by the DC component of the voltage source to a given level and then alternately charged and discharged with the AC component. In effect, the capacitor averages out the peaks and troughs of the AC component and, as a result, removes the AC ripple from the voltage source. This is known as the capacitor

decoupling the AC and DC components of the voltage source. This is a common technique used to remove electrical noise from power supplies, for example.

The flip side of this is that a capacitor can also be used to block the DC component of a voltage, allowing only the AC component to pass through (Figure 2-12).

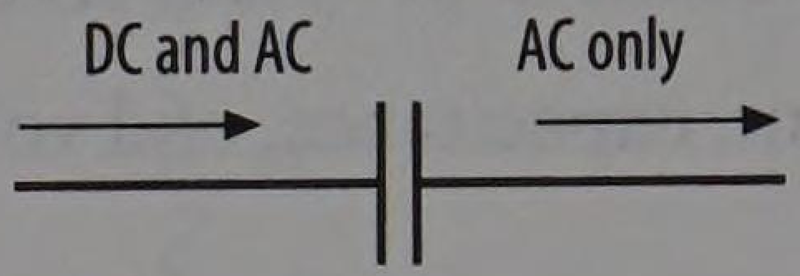


Figure 2-12. Blocking capacitor

Capacitors may also be used in series or parallel (Figure 2-13).

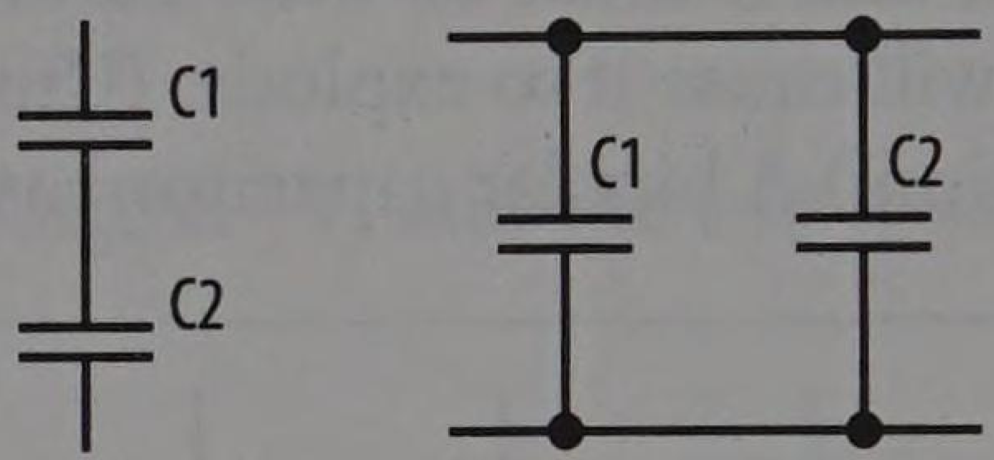


Figure 2-13. Capacitors in series and in parallel

The relationship is the opposite of what it was for resistors. In the series case, the total capacitance is calculated by:

$$C_{TOTAL} = C1 * C2 / (C1 + C2)$$

In the parallel case, the total capacitance is given by:

$$C_{TOTAL} = C1 + C2$$

Types of Capacitors

There are more than a dozen different types of capacitor, each based on a different technology. The ones you are most likely to come across are ceramic, electrolytic, and tantalum.

Ceramic capacitors are small in size and small in value. They range from a few picoFarads up to around 1µF. They are commonly used as decoupling capacitors for power-supply pins of integrated circuits and as bypass capacitors in crystal circuits (among other uses).

Electrolytics look like small cylinders and are used primarily for decoupling power supplies. They range in value from 100nF to several F (and we're talking *big* capacitors here). Their accuracy is terrible. Their actual value can vary quite a bit from what it is supposed to be. Therefore, they should not be used when critical tolerances are required. Use them only when ballpark values are sufficient.

The other problem with electrolytics is that they age, and the older they get, the worse they become. Expect a circuit using electrolytics to eventually fail. Having said

that, most consumer electronics still use them heavily, and for one reason—they are very cheap. By the time they've failed, the product will be well out of the warranty period. However, electrolytics will outlast the useful lifetime of your average computer product. You'll have upgraded your PC to a newer model long before its electrolytics have passed on.

> The most common cause of failure in old radios and hi-fi gear is that the electrolytics have failed. You can often pick up a very cheap bargain at a garage sale. Ten minutes with the soldering iron and you've replaced the electrolytics and what didn't work anymore suddenly comes back to life as good as new. Well, most of the time anyway.

Tantalum capacitors are somewhat larger than ceramics, but not as physically large as electrolytics. They range in value from around 100nF up to several hundred µF. They are commonly used to decouple power supplies. They are more accurate than electrolytics, meaning that their actual value is closer to their stated value. My company prefers to use tantalums over electrolytics in our designs whenever possible. We like our products to *last*.

RC Circuits

Combining resistors and capacitors can yield some interesting and useful effects. A resistor-capacitor combination is known as an *RC circuit*, and they can take one of three forms. In the first form, the resistor and capacitor are in parallel (Figure 2-14).

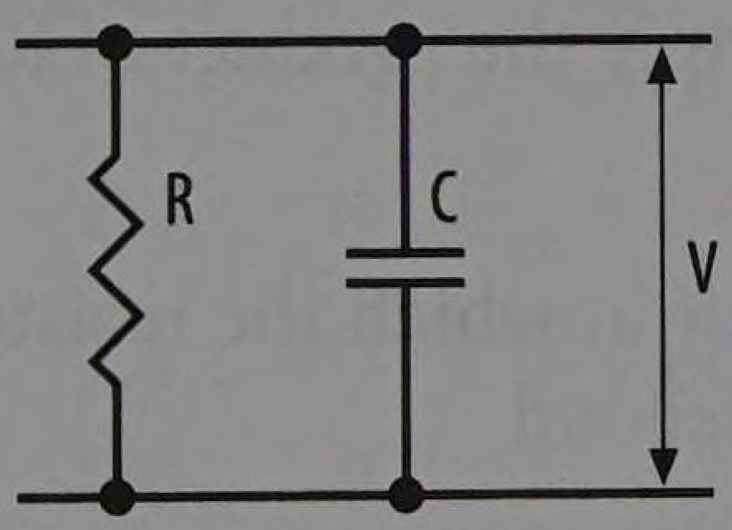


Figure 2-14. Resistor and capacitor in parallel

Now, what does this do? A voltage (V) applied across the pair will charge the capacitor (as well as some current flowing down through the resistor). When the applied voltage is removed, the capacitor will discharge through the resistor. The resistor will limit the rate of discharge, since it limits current flow. From Ohm's Law, we have that:

```
I = -V / R
```

(The negative voltage is because we're discharging the capacitor.) Now, the current flow out of a capacitor is given by:

```
I = C * dV/dt
```

So, we have:

```
dV/dt = -V / RC
```

Integrating this with respect to time, with zero initial conditions, gives us:

$$V = e^{-t/RC}$$

This gives us the discharge waveform shown in Figure 2-15, which represents the voltage across the capacitor.

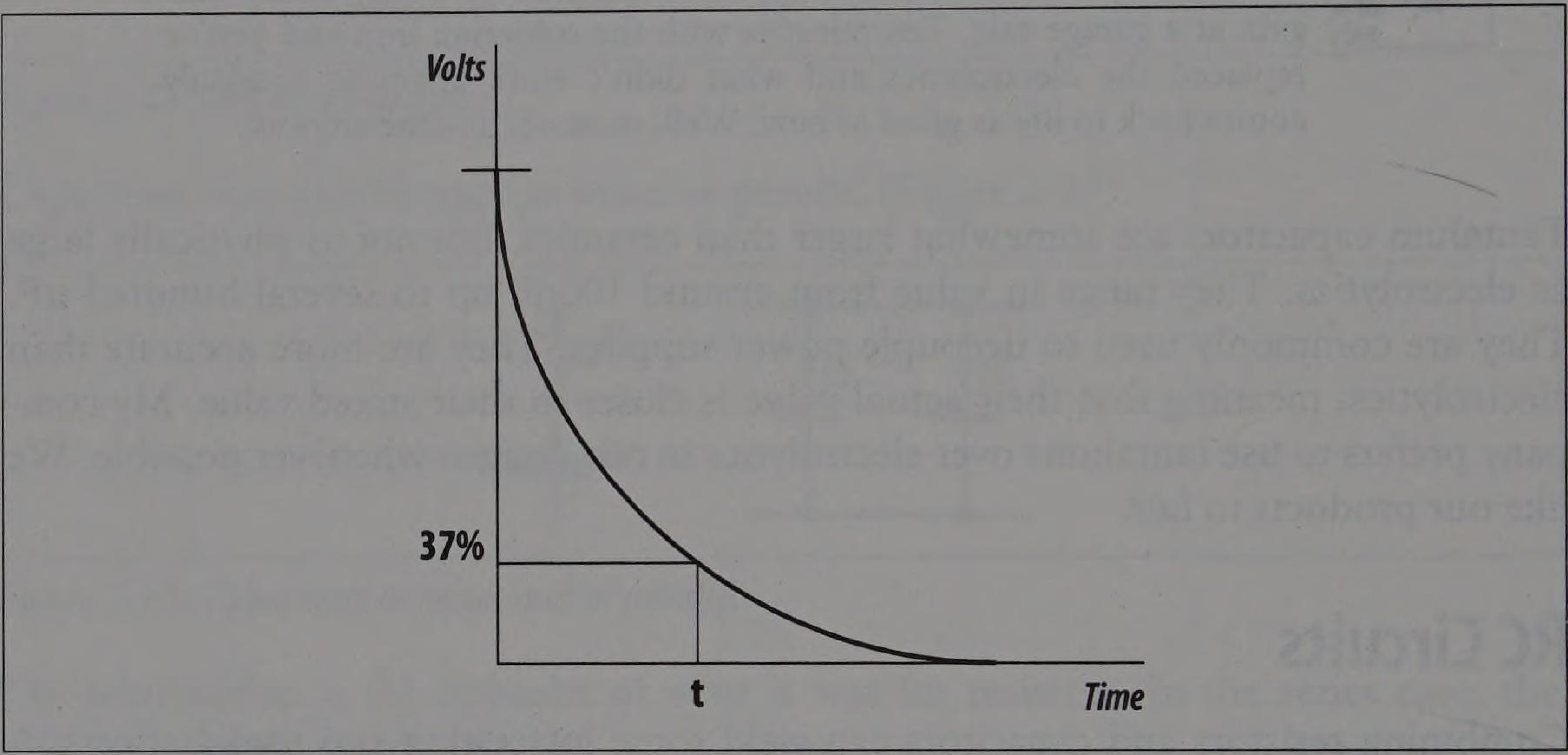


Figure 2-15. Discharge of a parallel RC circuit

A parallel RC circuit will provide an exponential decay in the output voltage. The value for t when the output voltage is at 37% of the maximum is known as the *time constant* for the circuit and is simply the product of R and C:

```
t = R * C
```

For example, a parallel RC circuit in which the resistor is 100kΩ and the capacitor is 10µF gives a time constant of 1 second.

The second form of RC circuit is the series RC circuit, shown in Figure 2-16.

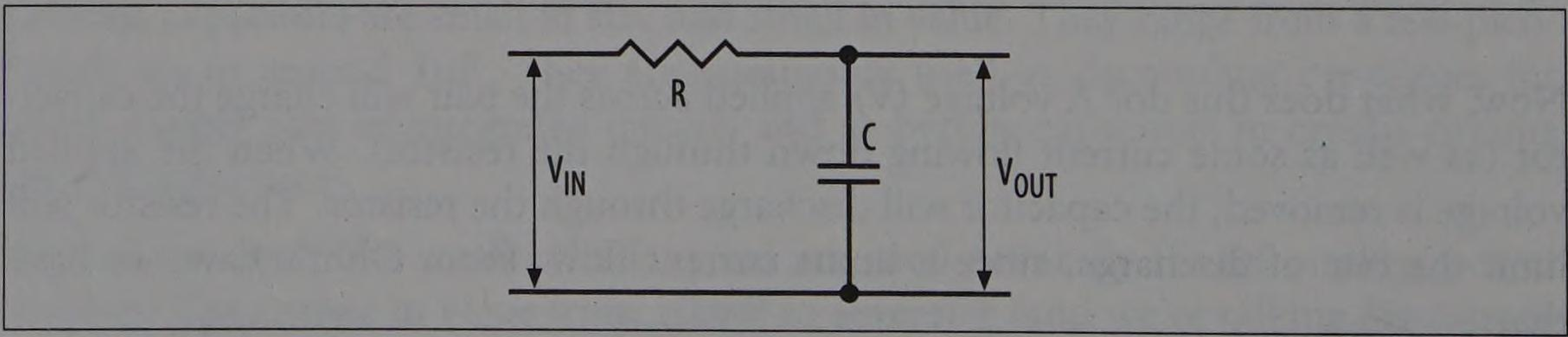


Figure 2-16. Series RC circuit

When a voltage is applied at the input to the RC circuit (on the left), current will flow through the resistor and the capacitor will begin to charge. However, the resistor limits current flow, and therefore limits the rate at which the capacitor charges.

Now, the current flowing into the capacitor is again given by the relation:

```
I = C * dV/dt
```

This current is the same as that flowing through the resistor, and by Ohm's Law, we have this current given by:

$$I = (V_{IN} - V_{OUT}) / R$$

where V_{IN} - V_{OUT} is the voltage drop across the resistor. Combining these two equations gives us the differential equation:

$$dV/dt = (V_{IN} - V_{OUT}) / RC$$

Integrating this gives us the voltage at the capacitor as:

$$V_{OUT} = V_{IN} (1 - e^{-t/RC})$$

Again, this is an exponential equation; however, this time, it represents an exponential charging of the capacitor. The waveform for the voltage at the capacitor is shown in Figure 2-17.

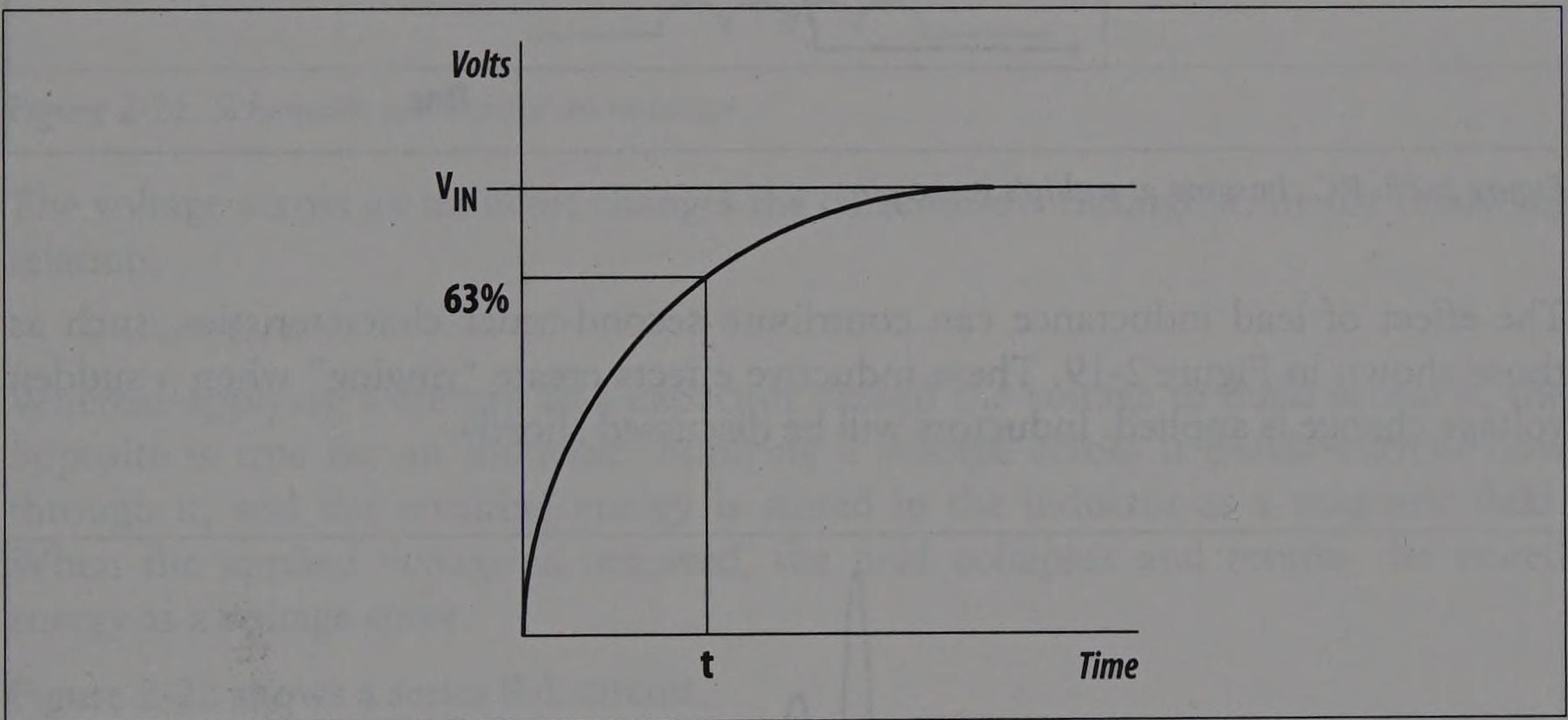


Figure 2-17. Charging of a series RC circuit

In this case, the time constant is the time for the voltage at the capacitor to reach 63% (total -37%) of the input voltage. As before, this time constant is simply the product of R and C.

This form of RC circuit is a simple type of *low-pass filter*. This is a circuit that provides a path to ground for high-frequency components of a signal, thereby attenuating them from the main signal, while the low-frequency components suffer far less attenuation. This type of circuit is very useful for removing high-frequency noise that may be superimposed on a signal.

A given processor or peripheral chip will have a small amount of *input capacitance* on each input pin. This, combined with the small inherent impedance of a circuit connection and the input impedance of the pin, means that an applied digital voltage to

the pin will actually appear as an exponential rise, rather than a sharp (digital) edge (Figure 2-18). These effects are minimal, but can be significant in high-speed circuits or when several devices are connected to the same signal line and the overall input capacitance is not insignificant.

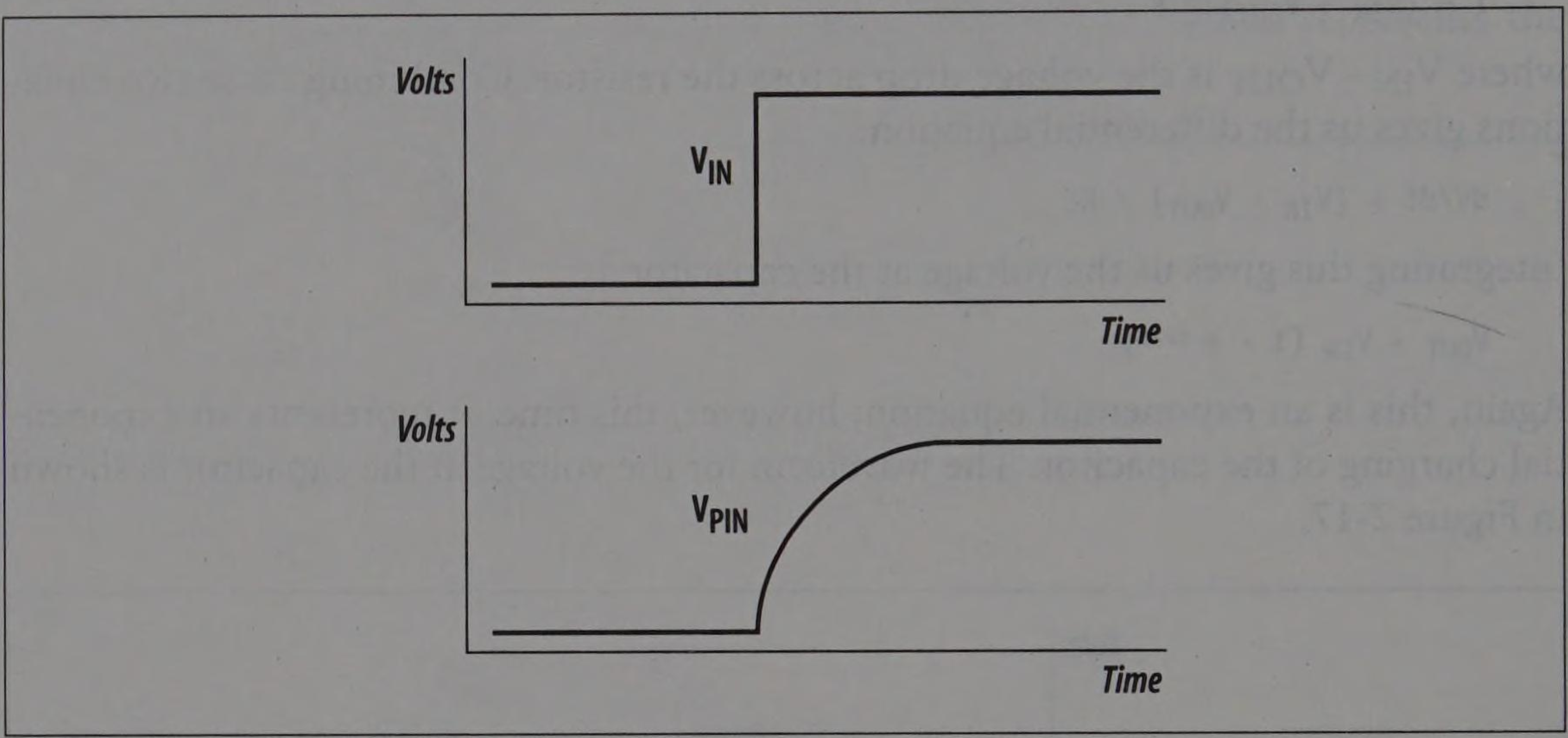


Figure 2-18. RC charging at a chip's input pin

The effect of lead inductance can contribute second-order characteristics, such as those shown in Figure 2-19. These inductive effects create "ringing" when a sudden voltage change is applied. Inductors will be discussed shortly.

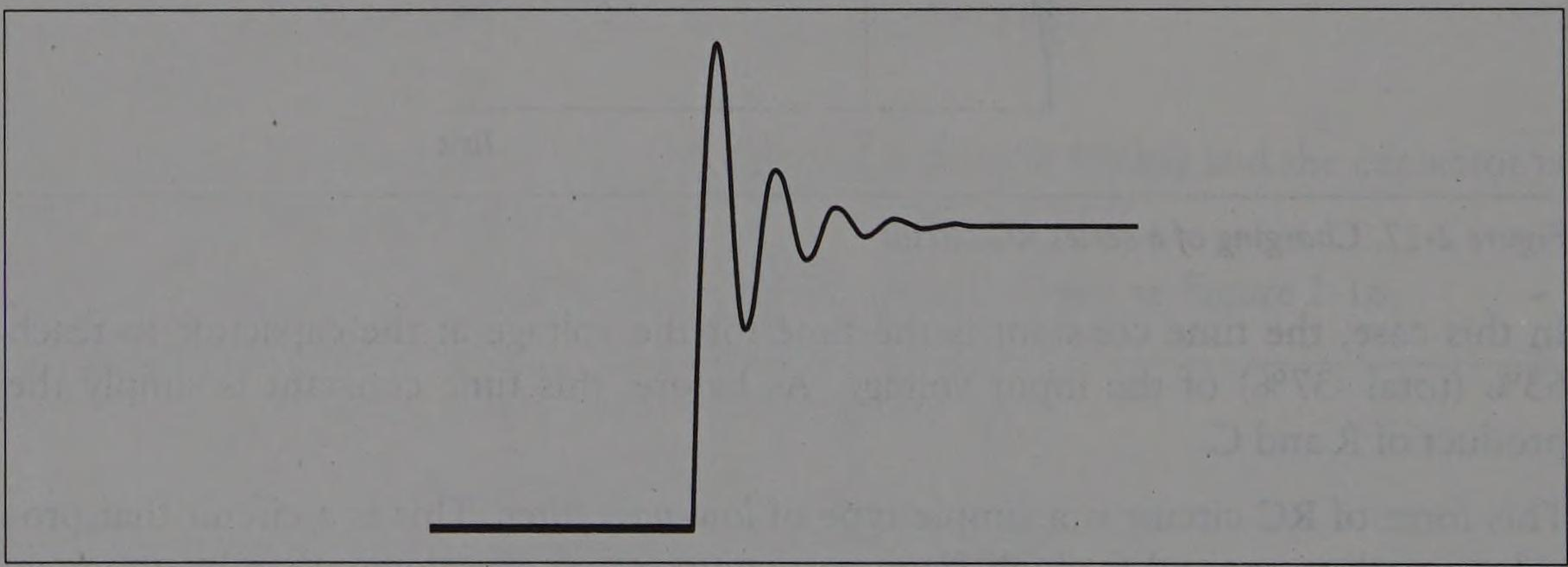


Figure 2-19. Inductive effects cause ringing on a signal input

The third form of an RC circuit is shown in Figure 2-20.

This type of circuit is a simple form of a *high-pass filter*, since it passes only the high frequencies through to the output. The capacitor in such a circuit is known as a *blocking capacitor*.

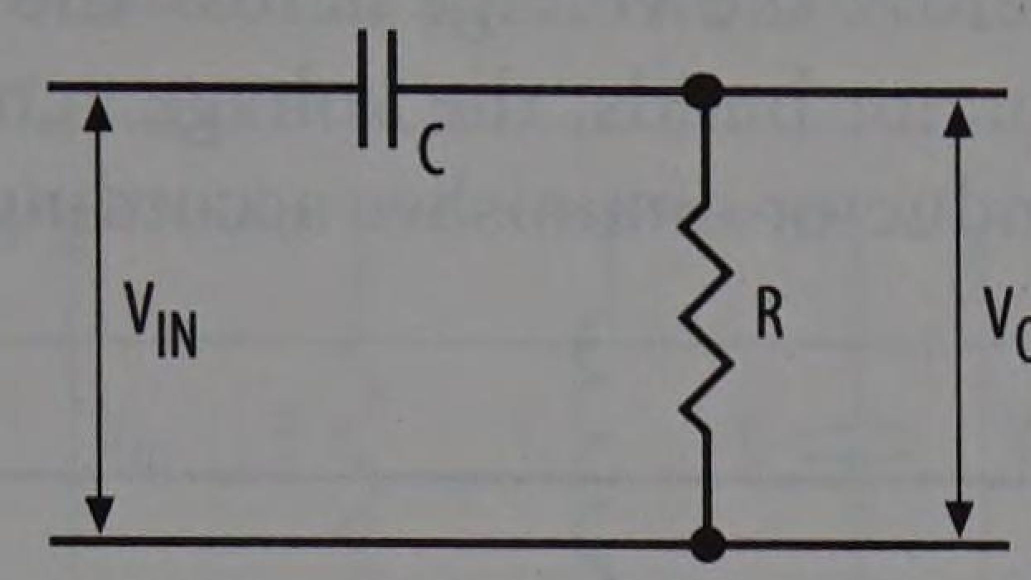


Figure 2-20. RC filter

Inductors

Inductors are passive components that are essentially a coil of conductive wire. The schematic symbol for an inductor is shown in Figure 2-21. Inductance is measured in *Henries*, with an equation symbol L and a unit symbol H.

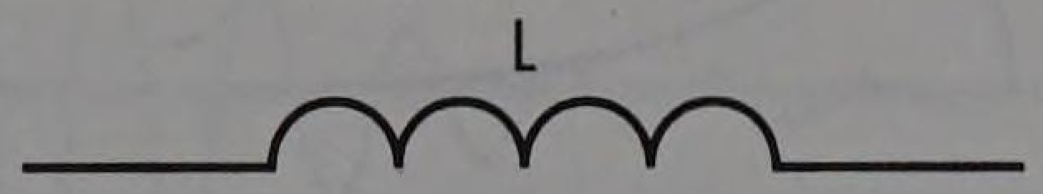


Figure 2-21. Schematic symbol for an inductor

The voltage across an inductor changes the current flow through it, by the following relation:

```
V = L * dI/dt
```

Whereas applying a current to a capacitor caused the voltage to build across it, the opposite is true for an inductor. Applying a voltage across it builds current flow through it, and the resulting energy is stored in the inductor as a magnetic field. When the applied voltage is removed, the field collapses and returns the stored energy as a voltage spike.

Figure 2-22 shows a series R-L circuit.

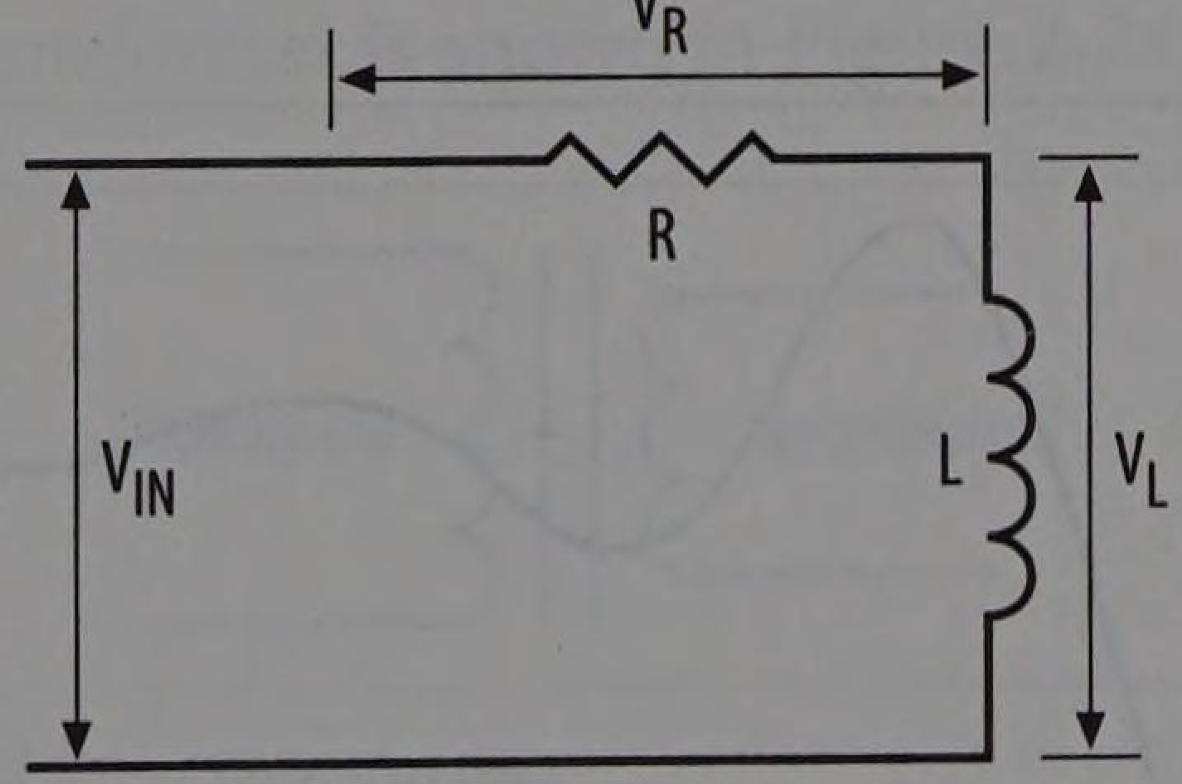


Figure 2-22. Series R-L circuit

The voltage across the resistor (V_R) and the voltage across the inductor (V_L) are shown in Figure 2-23. When a voltage is applied at V_{IN}, the voltage across the

resistor is initially small, whereas the voltage across the inductor is large. As the current flow through the inductor builds, the voltage across the resistor increases, while the voltage across the inductor diminishes accordingly.

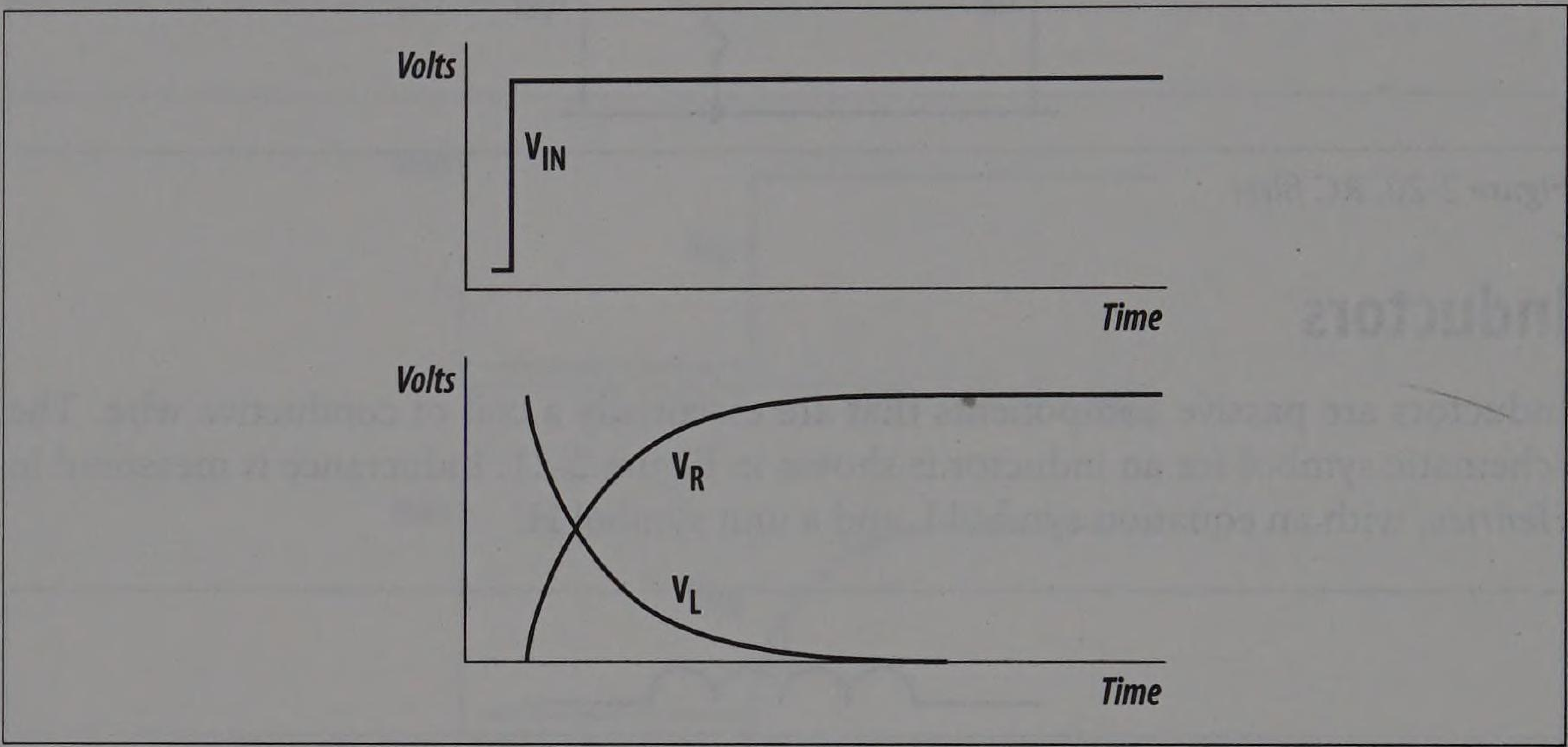


Figure 2-23. Series R-L response to a step input

Figure 2-24 shows a series R-L-C circuit.

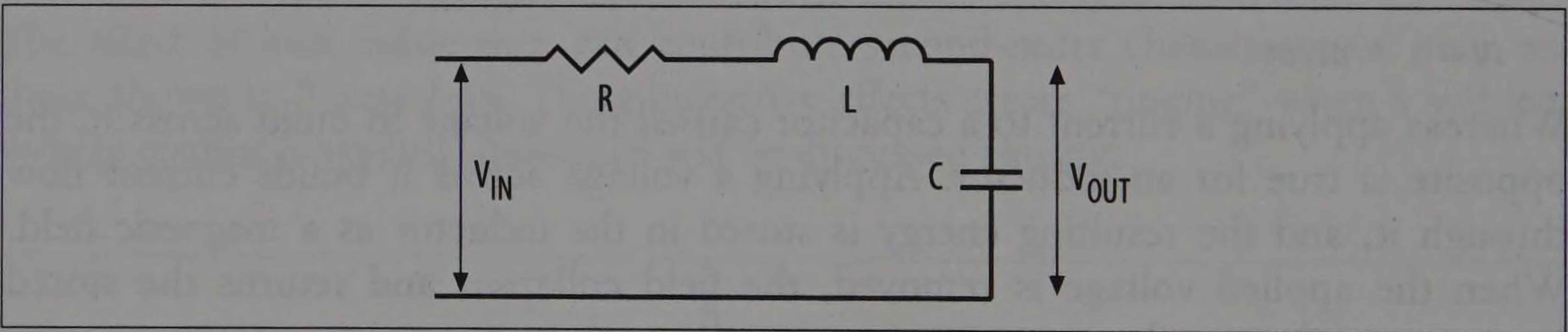


Figure 2-24. R-L-C circuit

The response (V_{OUT} versus time) of an R-L-C circuit to a step input is shown in Figure 2-25.

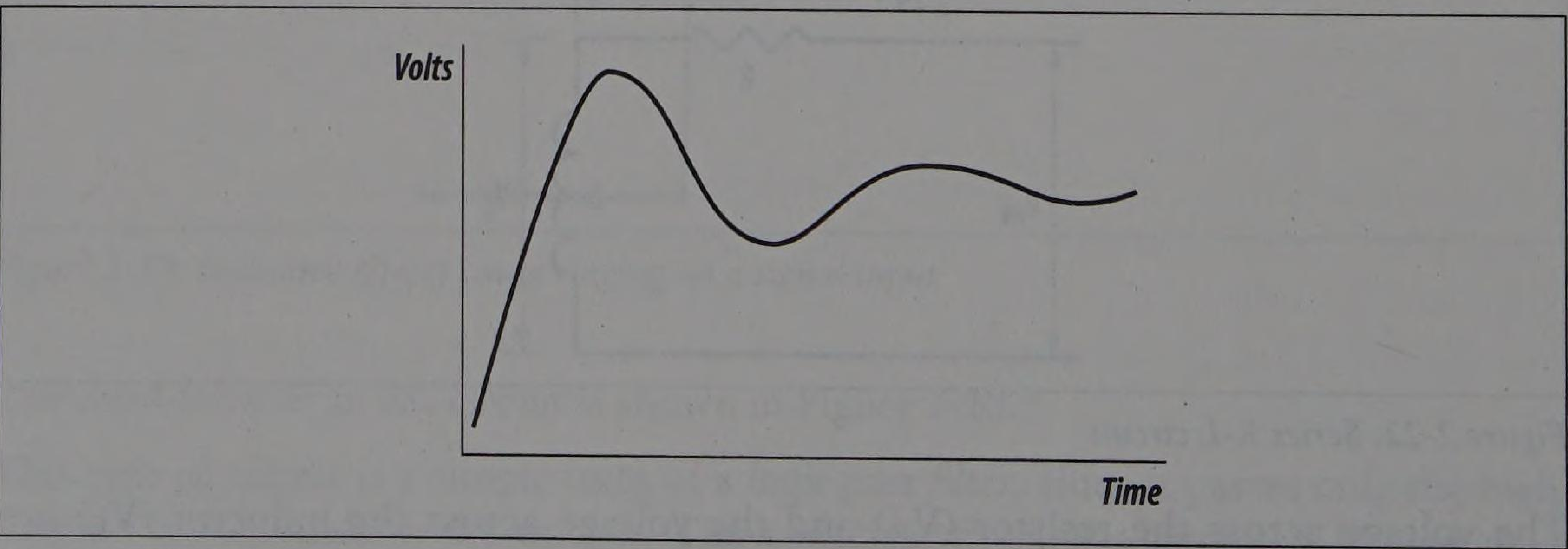


Figure 2-25. R-L-C circuit response

Figure 2-26 shows an R-L-C circuit in which all the components are in parallel.

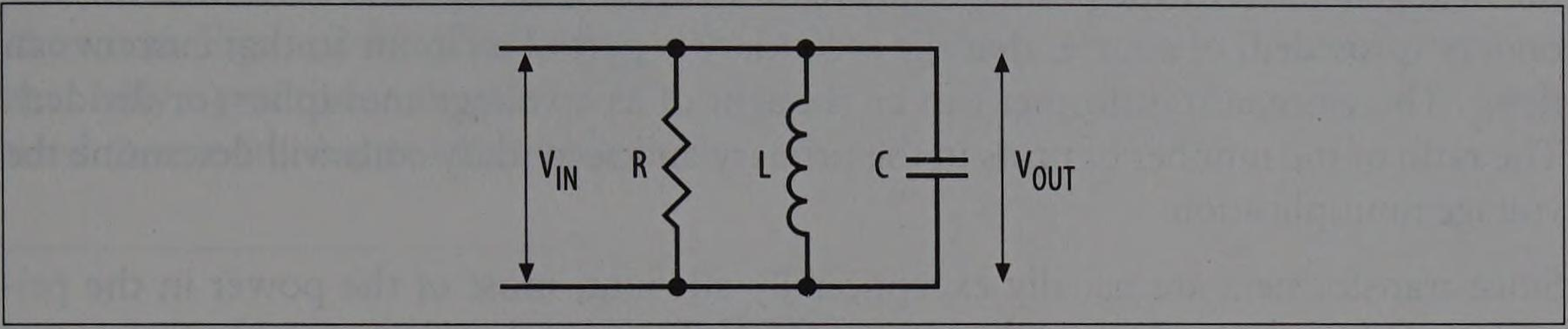


Figure 2-26. Parallel R-L-C circuit

The step response of this circuit is shown in Figure 2-27.

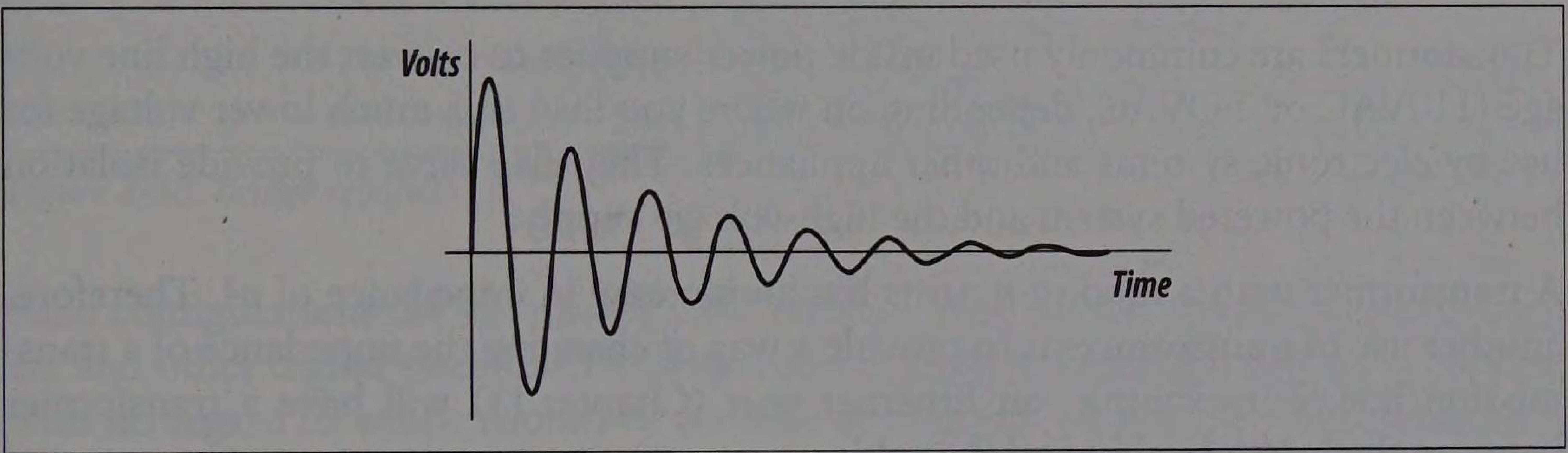


Figure 2-27. Parallel R-L-C circuit response

Inductors are commonly used in switching voltage regulators (Chapter 3) and are also employed (in combination with a resistor and capacitor) as filters to remove unwanted frequency components from a signal. Inductive effects exist in many components, and inductive voltage spikes are the bane of the embedded system designer.

Transformers

Transformers are related to inductors. A transformer consists of two coils of wire, known as the *primary* and the *secondary*, that are closely coupled magnetically. The schematic symbol for a transformer is shown in Figure 2-28.

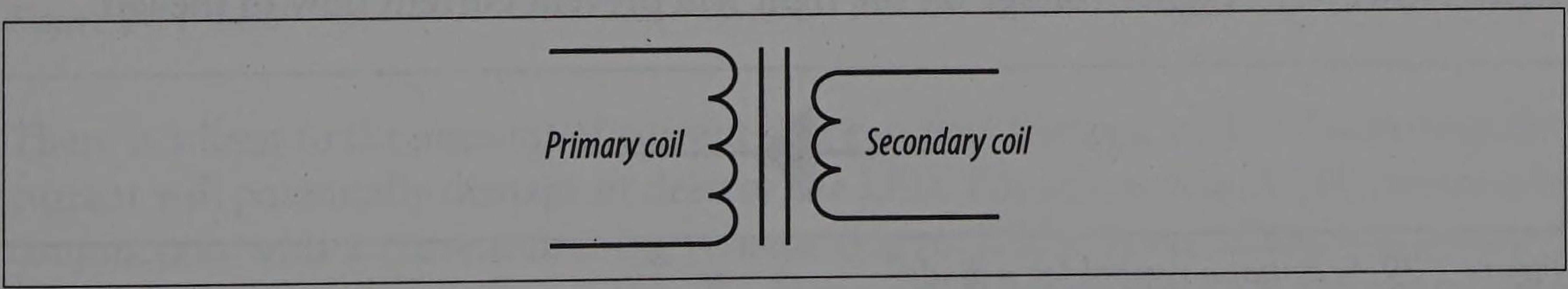


Figure 2-28. Schematic symbol for a transformer

An AC current flowing through the primary coil will generate an associated electromagnetic field. The strength of the field is proportional to the number of turns in the coil of the primary. Because the secondary coil is within this field, the field will generate a current flow through (and therefore a voltage difference across) the secondary.

Since the secondary has a different number of windings in the coil than the primary, the field generated by the primary will create a different voltage and current in the secondary (provided, of course, that the secondary is part of a circuit so that current can flow). Therefore, a transformer can be thought of as a voltage multiplier (or divider). The ratio of the number of turns in the primary and secondary coils will determine the voltage multiplication.

Since transformers are usually exceptionally efficient, most of the power in the primary is transferred across to the secondary. If the secondary increases the voltage of the primary, then the secondary's current will correspondingly be smaller than in the primary. Conversely, if the voltage across the secondary is less than the primary, the current through the secondary will therefore be larger than in the primary.

Transformers are commonly used inside power supplies to convert the high line voltage (110VAC or 240VAC, depending on where you live) to a much lower voltage for use by electronic systems and other appliances. They also serve to provide isolation between the powered system and the high-voltage supply.

A transformer with a ratio of n turns has an increase in impedance of n^2. Therefore, another use of transformers is to provide a way of changing the impedance of a transmission line. For example, an Ethernet port (Chapter 11) will have a transformer between the interface chip and the cable.

Diodes

Diodes are extremely useful semiconductor devices. They have the interesting characteristic that they will pass a current in one direction, but block it from the other. They can be used to allow currents to flow from one part of a circuit to another but prevent other currents from "backwashing" where you don't want them.

The schematic symbol for a diode is shown in Figure 2-29. The arrow indicates the direction of conduction. The arrow represents the *anode*, or positive side, of the diode, while the bar represents the *cathode*, or negative side, of the diode. A higher voltage on the left of the component will allow current to be passed through to the right. However, a higher voltage on the right will prevent current flow to the left.

Figure 2-29. Schematic symbol for a diode

Diodes have a *forward voltage drop* when conducting. This means that there will be a voltage difference between the anode and the cathode. For example, a diode may have a forward voltage drop of 0.7V. If this diode is part of a larger circuit and the voltage at the anode is 5V, then the voltage at the cathode will be 4.3V.

Diodes are useful for removing negative voltages from a signal, a process known as *rectification*. Four diodes may be combined together to form a bridge rectifier, as shown in Figure 2-30. The bridge "flips" the negative components of the wave so that only a positive voltage is present at the output. A capacitor on the output can be used to smooth the rectified wave.

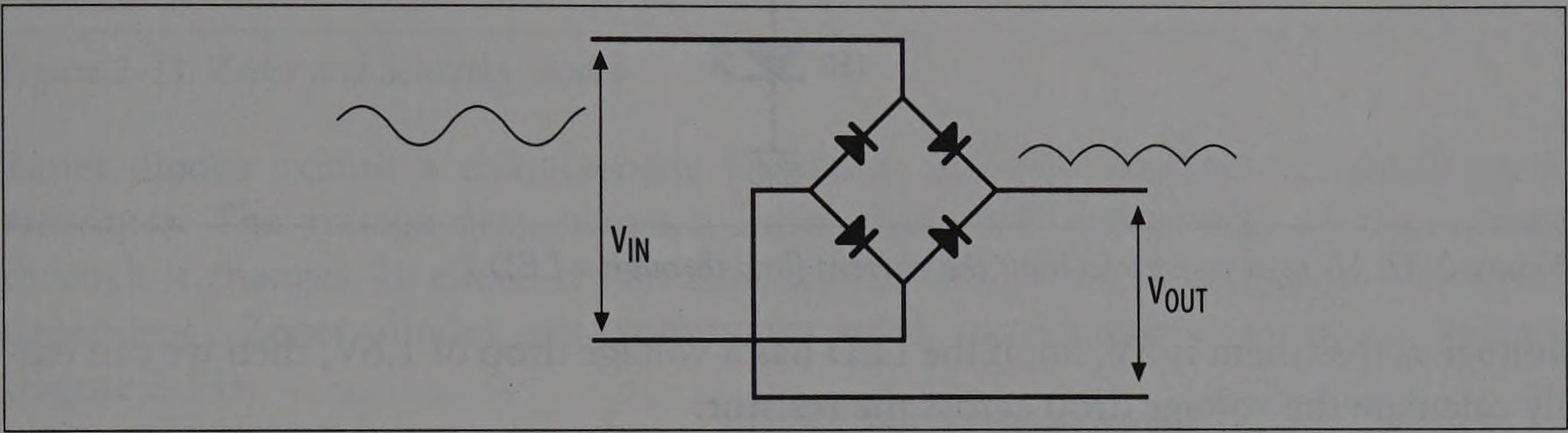


Figure 2-30. Bridge rectifier

Such configurations are commonly used on the power inputs to embedded computers and other digital systems. A voltage can be applied across the inputs on the left, with no regard to which should be positive or negative. The bridge rectifier ensures that a positive voltage will always be conducted to the upper right, and at the same time current flow is returned from the lower right, through the bridge rectifier to whichever lefthand connection is negative.

The most commonly seen diode is the LED (Light-Emitting Diode) (Figure 2-31). All diodes produce a small amount of light as a consequence of their operation (although you don't normally see it because of the diode casing), it's just that LEDs are especially good at it.

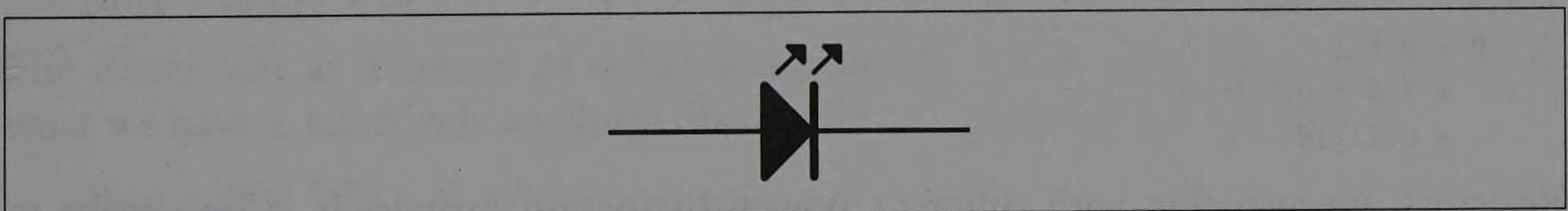


Figure 2-31. LED

There is a limit to the amount of current that can pass through a LED. Exceeding this current will potentially damage or destroy the LED. For this reason, LEDs are used in conjunction with a current-limiting resistor (Figure 2-32). Some LEDs will incorporate a current-limiting resistor internally. However, most do not, so it is important to check the manufacturer's datasheet. Generally, you'll need to include the resistor, and calculating the required value is easy.

Let's say that the LED has a forward voltage drop of 1.6V and a current limit of 36mA. We need to select a resistor that will limit the current flowing through the LED to this value. In our circuit, the LED and resistor are in series, and the total

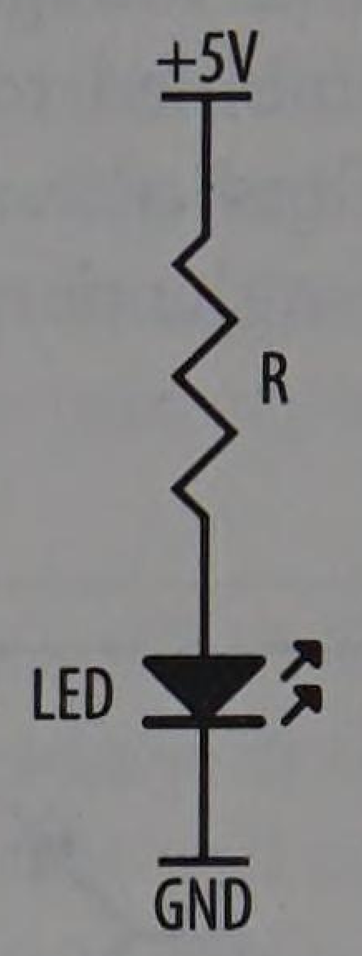


Figure 2-32. Using a resistor to limit the current flow through a LED

voltage across them is 5V. So, if the LED has a voltage drop of 1.6V, then we can easily calculate the voltage drop across the resistor:

$$V_R = 5 - 1.6 = 3.4V$$

So, if the voltage drop across the resistor is 3.4V and we need to limit the current to 36mA, using Ohm's Law, we can calculate a value for R:

$$R = V / I = 3.4 / 0.036 = 94.44\Omega$$

A 100Ω resistor will therefore do fine and will result in a brightly glowing LED. If you want lower intensity light, you just need to limit the current further, by using a larger resistor. Note that since 36mA is the maximum current the LED can handle, we will always need a resistor that keeps the current flow below this. Therefore, we always opt for a larger R.

The power that the resistor must dissipate is given by the relation:

$$P = V * I = 3.4 * 0.036 = 0.1224W$$

Resistors are available with different power-dissipation ratings. It is important to choose a resistor with the correct rating. In this instance, we would use a 0.125W resistor.

The ubiquitous power-on LED you see in your home appliances works in this exact way. This simple LED circuit (or variations of it) drives the LEDs on your PC's front panel, your VCR and DVD player, your cell phone, and a host of other appliances. Many traffic lights and railroad signals are replacing conventional bulbs with arrays of LEDs, as the LEDs last longer and produce more light (per area).

LEDs are available in red, green, yellow, blue, and white. The last two colors are very hard to produce and therefore expensive to buy.

In Chapter 6, we'll see how to control a LED using a microprocessor.

An understanding of two more types of diodes may be useful. They are Zener diodes and Schottky diodes (Figure 2-33).

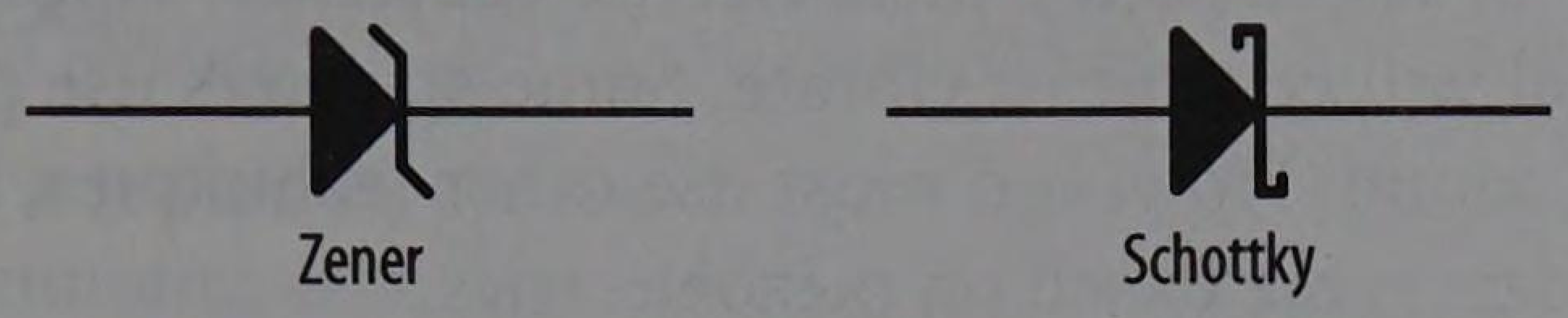


Figure 2-33. Zener and Schottky diodes

Zener diodes exhibit a characteristic known as *dynamic resistance* or *small-signal resistance*. The voltage drop across a Zener diode will not change as the current through it changes. In effect, it acts as a variable resistor whose resistance is current dependent. Zener diodes are commonly used to provide a reference voltage (Figure 2-34).

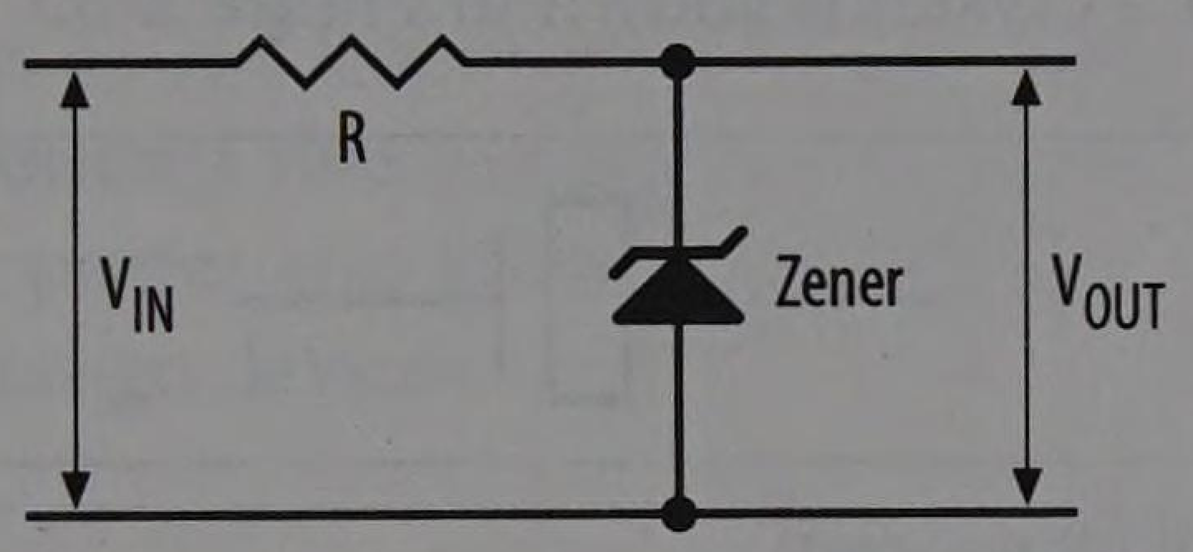


Figure 2-34. Using a Zener diode to provide a reference voltage

From Ohm's Law, we have that:

$$(V_{IN} - V_{OUT}) = I * R$$

Now, if V_{IN} changes, it logically follows that the current will also change. So we can modify our equation thus:

$$(\Delta V_{IN} - \Delta V_{OUT}) = \Delta I * R$$

The Zener acts as a source of dynamic resistance, which we'll designate R_d. Now what we have is effectively a voltage divider. So, our equation for V_{OUT} is:

$$\Delta V_{OUT} = \Delta V_{IN} * R_d / (R + R_d)$$

Schottky diodes are also known as *hot-carrier diodes* and behave like conventional diodes, save for a very small forward voltage drop. They are commonly used in power-supply circuits and signal rectification for this reason.

Crystals

Finally in our component tour, we come to crystals. Just as their name suggests, they are a small block of quartz (silicon dioxide). Quartz crystal is a type of material known as a *piezoelectric*. This is a substance that generates a voltage when it is stressed (compressed, stretched, twisted). This effect is utilized in microphones. The sound vibrates the piezoelectric material, and it produces a small AC voltage that is

directly proportional to the original sound that created it. This voltage is then amplified for broadcasting, recording, or processing.

The opposite effect is also true for piezoelectric materials. Apply a voltage and the piezoelectric material will contort or vibrate. Some speakers use piezoelectric materials to produce their sound. However, most use other techniques, such as electromagnetics. Most loud buzzers are based on piezoelectrics.

Now, the neat thing about quartz is that for a block of a given size, it will vibrate at a given (and fixed) frequency. For that reason, it can be used as an oscillator to generate a sine wave, which in turn can be used to generate timing signals for microprocessors and other digital circuits. Just about every computer system will have a crystal (or two) somewhere on its circuit board, generating the timing that ultimately drives the whole machine. That crystal is simply a small block of quartz, plated at either end with wires attached and encased in a metal can.

The schematic symbol for a crystal is shown in Figure 2-35.

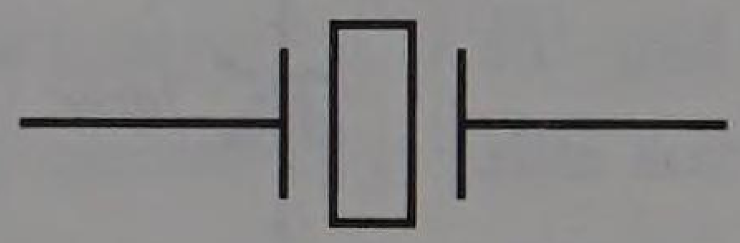


Figure 2-35. Schematic symbol for a crystal

Crystals require a drive circuit to make them go. These tend to be a bit temperamental and don't always oscillate at the frequency you expect, due to a range of effects that are hard to track down. Fortunately, there are two easy ways around this problem. The first is that most processors (and other chips requiring timing) have internal oscillator circuits. All you need to do is add the external crystal (and maybe a capacitor or two), and it will work beautifully. For chips that don't make life quite so easy, you can get complete oscillator modules, which include the crystal and drive circuit. All you need to do is give them power and ground, and they too work beautifully.

Clocks and Oscillators

All microprocessors (and quite a few other digital devices too) require clocks. A *clock* is an output from an oscillator that runs the processor, and all system events relate to the clock. (And just in case you're wondering, this clock has nothing to do with the time of day. Think of it as a stream of digital pulses.) The clock frequency is normally expressed in kiloHertz (kHz), MegaHertz (MHz, 1000kHz), or GigaHertz (GHz, 1000MHz). The clock frequency of a processor is also known as its *clock speed*.

A given processor will have a maximum and a minimum clock frequency. This specifies the range in which the oscillator driving the processor can operate. A processor with a minimum clock speed of zero is said to have *static operation* or *DC operation*. This means that the processor can have its clock stopped and still be able to resume

operation at a later time with no ill effect. If the minimum operating frequency of a processor is greater than zero, then the processor is said to have *dynamic operation*. If the oscillator frequency falls below the minimum of a dynamic processor, then that processor may suffer corruption of its register content.

The clock speed of a processor relates to how quickly a processor can execute software. A processor running at a faster clock speed will execute software faster than a processor *of the same type* running at a slower clock speed. But clock speed is not the whole story in terms of processor speed. One processor architecture may take 32 clock cycles to execute an instruction, whereas another processor may complete one instruction every clock cycle. So, even though these two processors are running at the same clock speed, the latter will be significantly faster than the former.

There are several ways of generating a clock. Which is appropriate depends largely on the processor you are using. Some processors expect a digital (square-wave) clock input. For a processor running at common frequencies, and this includes most processors, the best choice is to use a device called an *oscillator module* (Figure 2-36). These are four-pin components that provide a square-wave clock output at a given frequency, requiring only power and ground connections. These simplify the system design, as they are plug-and-go devices.

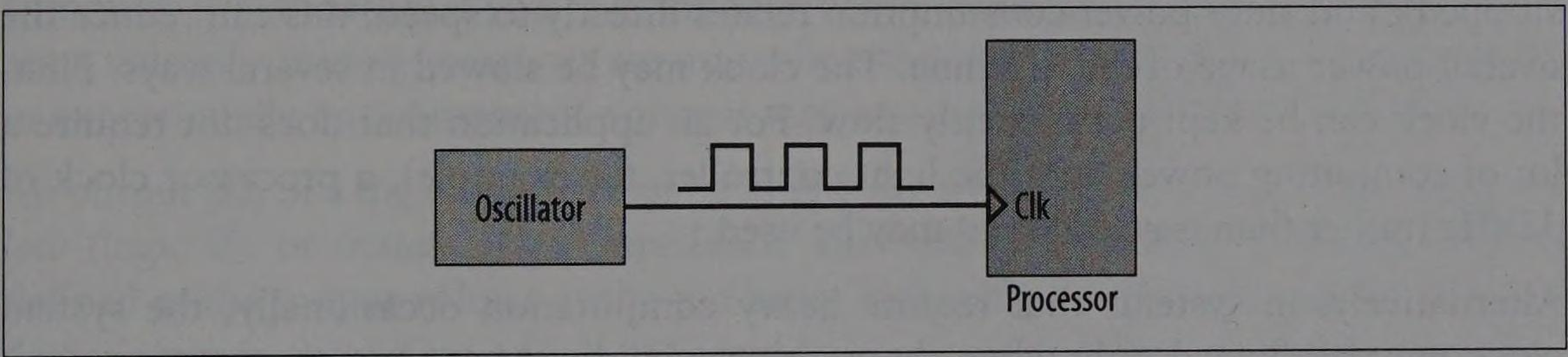


Figure 2-36. Microprocessor oscillator module

Many processors (including all the microcontrollers I can think of) contain oscillator circuitry and generally require only the addition of an external crystal and bypass capacitors (Figure 2-37). The capacitors remove higher-order harmonics from the oscillation.

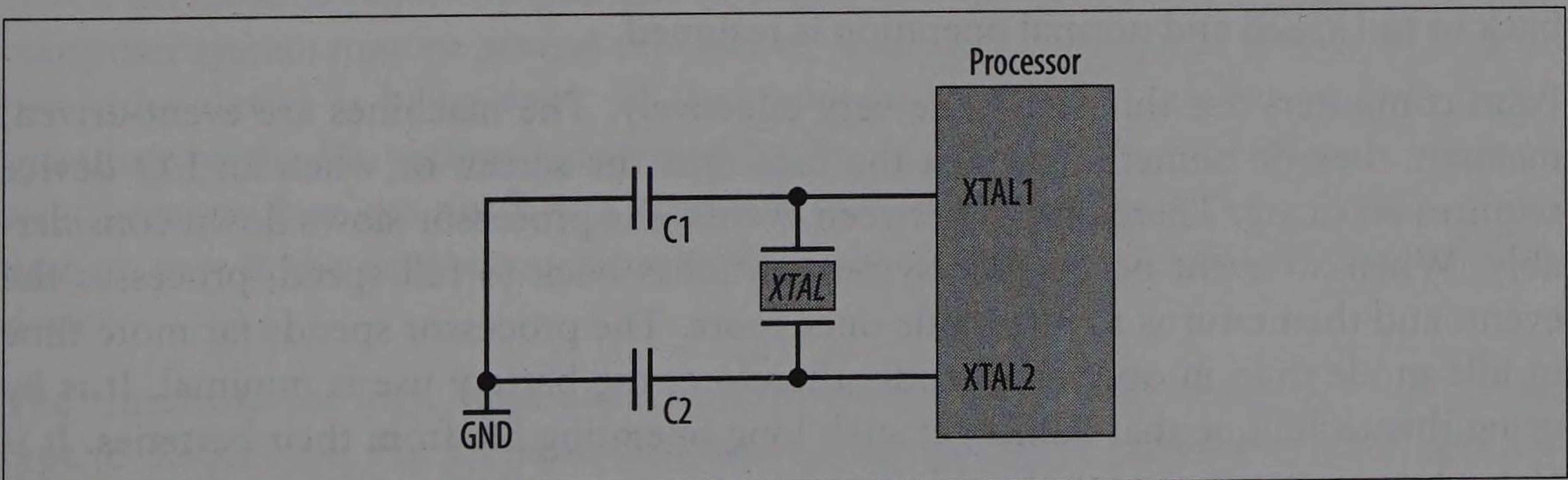


Figure 2-37. Crystal circuit for an internal oscillator

Power versus speed

Often it is necessary to design a system with minimal power consumption. This may be done to reduce the heat it produces or to make it portable. The more current the devices within a system use, the hotter they become. Too much heat will cause them to stop working. Although cooling subsystems and temperature monitors can be added, the better approach is to simply reduce the power consumption and therefore the heat. When a system is powered by batteries (or hamsters), the lower its current draw, and the longer the batteries (or hamsters) will last. For these reasons, low-powered design is advantageous.

Most of the current usage of a digital system occurs during transitions of state, in other words, during the clock edges. The more frequent the clock edges, the more the average current usage goes up. Therefore, the faster a processor runs, the more power it consumes. Conversely, the lower the processor's operating frequency, the less its power consumption. Many embedded processors are available in lower power versions. Power consumption can also vary between architectures. An ARM processor running at the same clock speed and operating voltage as a Pentium will have *considerably* lower power consumption. It is for this reason that ARMs are common in PDA devices.

Using devices with static operation allows the clock of the system to be slowed (or stopped), and since power consumption relates directly to speed, this can reduce the overall power usage of the machine. The clock may be slowed in several ways. First, the clock can be kept permanently slow. For an application that does not require a lot of computing power (a traffic light controller, for example), a processor clock of 32kHz (rather than, say, 20MHz) may be used.

Alternatively, in systems that require heavy computation occasionally, the system clock may be slowed only when the processor is idle. Many laptop computers use this technique to reduce their power consumption. The processor runs at full speed when in use. If the system sits idle for 30 seconds, the clock is slowed down into the kHz range. If the system remains idle for several minutes, the clock is slowed down into the Hz range. Since the user is not actively working with the machine (that's why it was idle), the user doesn't notice any difference. The moment the processor is required to perform a task (for example, if a key is pressed), the clock is switched back to full speed and normal operation is resumed.

Palm computers use this technique very effectively. The machines are event-driven, meaning they do something when the user taps the screen or when an I/O device requires servicing. Therefore, in between events, the processor slows down considerably. When an event occurs, the system switches back to full speed, processes the event, and then returns to idle mode once more. The processor spends far more time in idle mode than in operating mode; therefore, the battery use is minimal. It is by using this technique that Palms get such long operating life from their batteries. It is also why you never see (or shouldn't see) computationally intensive applications on Palm computers. The batteries would be dead in no time.

Some computer systems may even halt the clock completely until the processor is required, at which point an external device reactivates the system clock.

Many processors have SLEEP and HALT modes, reducing the processor's power consumption. Some processors extend this to SLEEP, NAP, DOZE, SNOOZE, and so on, each with a different level of power usage and each requiring a different period of time for the processor to "awaken." (The deeper the sleep, the longer it takes for the processor to resume operation.)

That concludes the discussion of electronic components. One major type of component that I haven't covered is the transistor. It has been left out simply because they are not commonly seen in embedded systems, and a proper coverage of transistors would occupy a large volume in its own right. If you are interested in learning about transistors, plenty of excellent books are available; just visit your local technical bookstore or cruise the Internet.

Digital Signals

Being an electronic circuit, the operation of a computer is about voltages and current flow. Understanding the basic principles of voltages and current flow within the computer is mandatory if you're going to produce a working system. Common operating voltages inside a computer are normally either 5V or 3.3V. For some low-power or exceptionally fast computers, voltages may be as small as 1.8V or even lower.

An output pin of a digital device can be in one of three states. It can be *high* (logic 1), *low* (logic 0), or *tristate* (*high impedance*, also known as *floating*). A logic high is defined as the output voltage at the pin being higher than a given threshold. When a device's pin is outputting a high, it is said to be *sourcing current* to that connection. Similarly, a logic low is when the output voltage is below a given threshold, and the device's pin is said to be *sinking current*. Typically, components can sink more current than they can source.

A tristate pin is outputting neither a high nor a low. Instead, it becomes high impedance (high resistance) so that current flow in or out of the pin is negligible. It is, in effect, invisible to other components to which it is connected. For example, within a computer system may be several memory devices connected to the data bus. When a particular device is being read, its data outputs will be either high or low (corresponding to the bit pattern being read back). All other memory devices in the system, because they are not being accessed, will have their data buses tristate. They take no part in the read transaction between the processor and the accessed memory device.

The threshold for logic high and the threshold for logic low can vary from device type to device type. For an input device to recognize a given signal as high or low, the output device must provide that signal within the appropriate limits. The thresholds can vary, but are always consistent across devices of the same *logic families*. Back in

the good old days, the number of logic families was limited, and each device within a family conformed to the thresholds of that family. Life, and designing digital systems, was easier. Now, with the quest for ever-lower-powered devices and the desire for devices to be as versatile as possible, there is considerable diversity with the thresholds for logic high and logic low. So the input low threshold for a given chip may not match the output low threshold for the chip to which it is connected. Therefore, it is vitally important to check the datasheets of all the components you are using and ensure that they will work together.

Voltage Thresholds

The voltages for the TTL family (Transistor-Transistor Logic) are defined as a logic low as a maximum input voltage of 0.8V and a maximum output voltage of 0.4V; and a logic input high as a minimum voltage of 2.0V and an output high as a minimum voltage of 2.4V. Many processors accept TTL inputs, though relatively few of them are actually TTL-compatible devices. This is important. You can never assume that a given output or a given input will be within voltage specifications. For instance, a *minimum* high voltage for an old 80386 processor on its clock input is 4.2V at 20MHz and 3.7V at 25MHz. These are significantly higher than standard TTL levels. A standard TTL device driving this input may not be able to achieve a voltage sufficiently large enough to be recognized as a high by this processor. The moral of the story is *check the datasheet*! The electrical (and timing) specifications are listed in datasheets for very good reasons.

There are many different logic families, each with its own threshold voltages and other characteristics. Beyond that, there are many components that don't fit into any particular logic family. The component datasheets are your best guide as to what will work with what.

When a device outputs a logic high and its output voltage is greater than the high threshold for the input device, current will flow from the output pin to the input pin. The output device is sourcing current, while the input device is sinking current.

Conversely, for certain types of digital logic, when a device outputs a logic low and its output voltage is lower than the low threshold of the input device, current will flow from the *input* pin to the *output* pin, even though the output device is the one controlling the voltage. The output device is sinking current, while the input device is sourcing current (Figure 2-38).

The magnitude of the current flow is important. A given device will have limitations on how much current it can sink or source. Exceeding this current limit can permanently damage an integrated circuit. It is therefore important to calculate the current flows within your system and ensure that all the requirements are met.

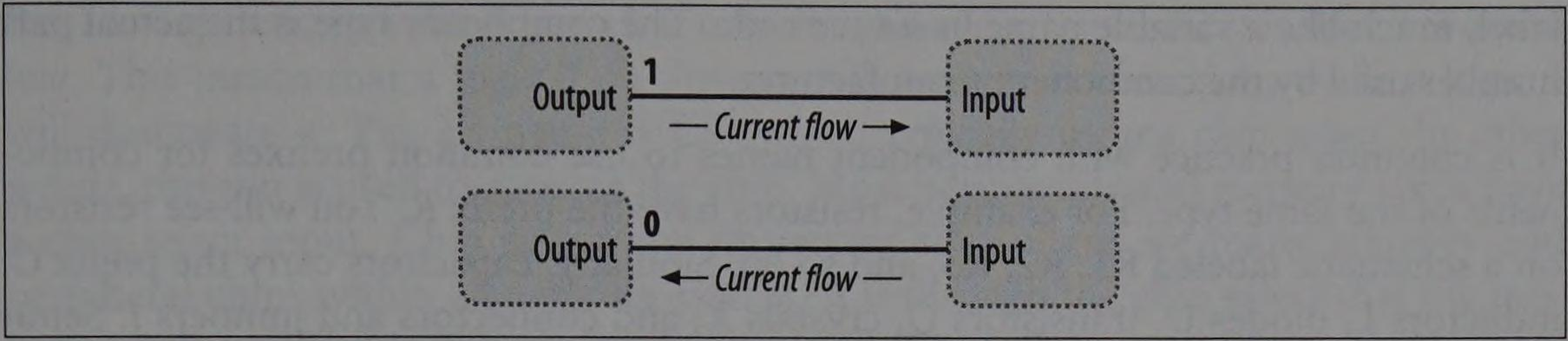


Figure 2-38. Current flow between digital devices

Understanding Schematics

You won't get very far in electronics unless you know how to draw and read schematics. They crop up everywhere, and understanding them is a must. The schematics are like an architect's blueprint. They show what components will be used in a circuit and how they are connected together. The schematics may also include other information such as construction directives. A schematic may have a list of revisions indicating what changes have been made to the original design. These are commonly called *Engineering Change Orders*, or *ECOs* for short. As a design grows and changes over time, it's a good idea to keep track of what changes were made and, just as important, *why* they were made. Just as commenting source code is important, so is keeping track of the ECOs.

You will come across two types of schematics. You will see schematics in datasheets, books like this one, and other technical documents. These schematics will just show the circuit (or partial circuit), maybe a note or two, and that's all. The other sort of schematic is the actual drawing(s) used to generate a circuit board. These schematics represent a full system design and will often have a title block located in the lower right of the sheet, indicating what the sheet represents, as well as who drew it and when. Figure 2-39 shows an example title block.

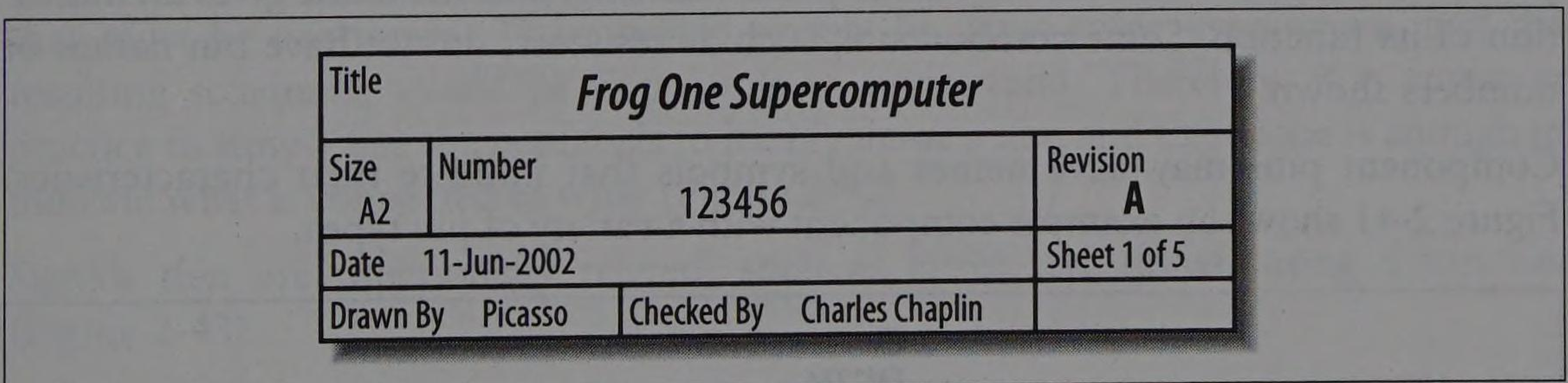


Figure 2-39. Title block

Essentially there are two types of objects on a schematic, components and nets. *Nets* are the wires that show what is connected to what. A component will have a component name and a component type. For example, a memory chip may have the name U3 and have a component type AT45DB161. The component name is simply a reference

label, much like a variable name in source code. The component type is the actual part number used by the component manufacturer.

It is common practice with component names to use common prefixes for components of the same type. For example, resistors have the prefix *R*. You will see resistors on a schematic labeled R1, R2, R3, and so on. Similarly, capacitors carry the prefix *C*, inductors *L*, diodes *D*, transistors *Q*, crystals *X*, and connectors and jumpers *J*. Semiconductors often carry the prefix *U*, but not always. Logic gates and other small, nondescript semiconductors may have the prefix *U*, but larger semiconductors may have a more informative name. For example, a processor may be labeled PROC while four memory chips may carry the names RAM0, RAM1, RAM2, and RAM3. Giving larger devices more meaningful names often makes schematics easier to understand. However, that said, a lot of people still give every semiconductor the *U* prefix.

Figure 2-40 shows an example component with a net.

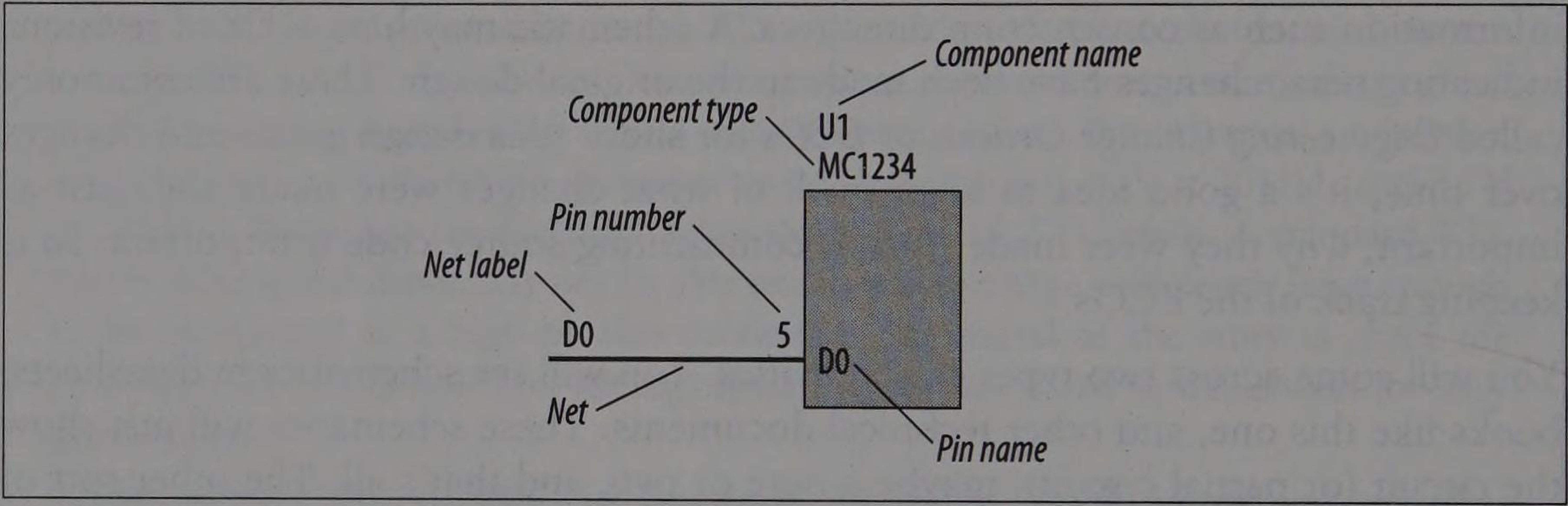


Figure 2-40. Signal net and component

As well as the name and part number, the component will also have an array of pins. The pins may have a number, a name, or both. The number indicates the physical pin on the chip to which the schematic pin is referring, and the name gives an indication of its function. Some components, such as resistors, do not have pin names or numbers shown.

Component pins may have names and symbols that indicate their characteristics. Figure 2-41 shows an example component with a variety of pin types.

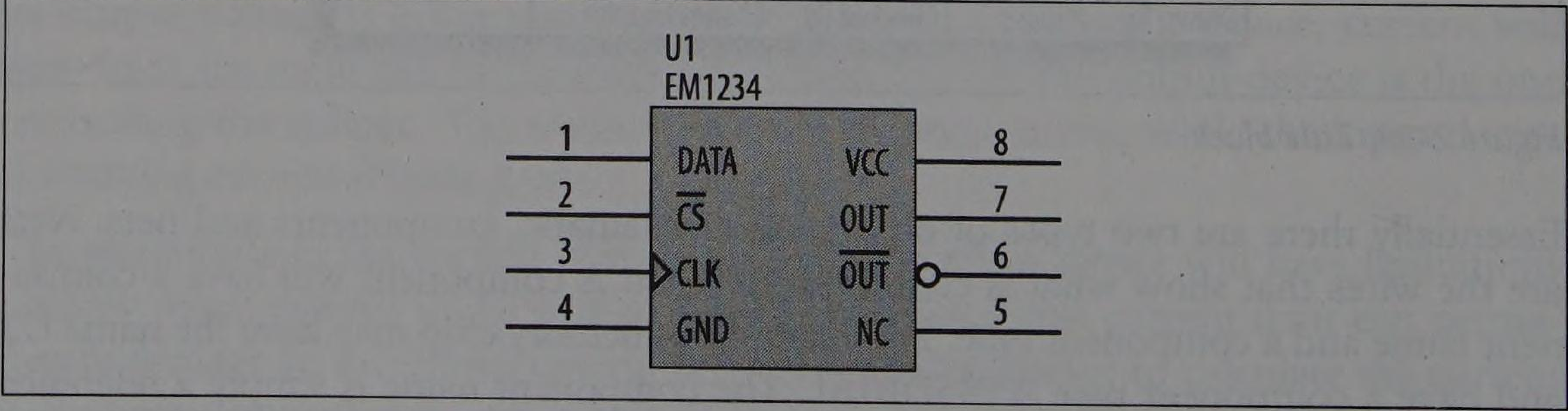


Figure 2-41. Pin types

Pin 1 is a generic pin. Pin 2 has a bar over the pin name that indicates that it is *active low*. This means that a logic 0 on this pin will activate its function, while a logic 1 will deactivate it. Pin 2's name is $\overline{\mathbf{CS}}$, which typically means *chip select*. In other words, this pin is used to activate the chip. Most peripherals and memory chips have a chip select input. Chip selects are important since there are many memory and peripheral chips within a computer system. It is through the chip select that the processor will enable the chip so that it can write data to it or read data from it. Some devices have an input called $\overline{\mathbf{CE}}$, which means *chip enable*. It's exactly the same as a chip select. They are just two different names for the same function.

The little triangle on pin 3 indicates that it is an edge-triggered input.

Pins 4 and 8 are ground (**GND**) and power (**VCC**), respectively. *VCC* and also *VDD* are used to label voltage sources for powering the circuits. The terminology originates from transistors and solid-state electronics, in which "collectors" (VCC) and "drains" (VDD) are common parlance. You don't need to worry about what the names mean, just know that when you see VCC or VDD, they relate to supply voltages.

Pin 7 is an output that is active high, and pin 6 is an output that is active low. Note the circle on pin 6. This indicates that it is an inverted output. That it has the same name as pin 7 indicates that pin 6 is the inversion of the output of pin 7. Finally, pin 5 is labeled NC. This is commonly used to represent "No Connect," which means that this pin has no function. No net should be connected to it. (Very rarely you'll also see a pin named "Do Not Wire." It means the same thing.) However, just seeing a pin named NC doesn't mean that you should *assume* that it is a no connect. It may just be that the chip manufacturer labeled the pin NC for some other reason. As always, check the datasheet carefully for each device.

A net may be drawn between two components, or a net may simply have a *net label* giving the net a name and indicating that it is connected to every other net with the same name. With complicated schematics, it may not be practical to show every wire that must be connected. There would simply be wires going everywhere, and the resulting schematic would be impossible to understand. Therefore, it is common practice to simply use the net labels to locally name a net, and this alone is enough to indicate what is connected to what (Figure 2-42).

Signals that are functionally related, such as buses, are drawn using a bus net (Figure 2-43).

A design often employs more than one schematic sheet. Just as a program is broken up into functions, with commonly used code placed in libraries, so too are designs broken into functional units, allowing subsystem reuse in multiple designs. For example, the same power-supply circuit may be used in several different embedded computer designs. By placing the power-supply circuit on its own sheet, that same subsystem design may be reused in many designs. *Ports* are used to indicate when a schematic's nets are connected to another schematic sheet.

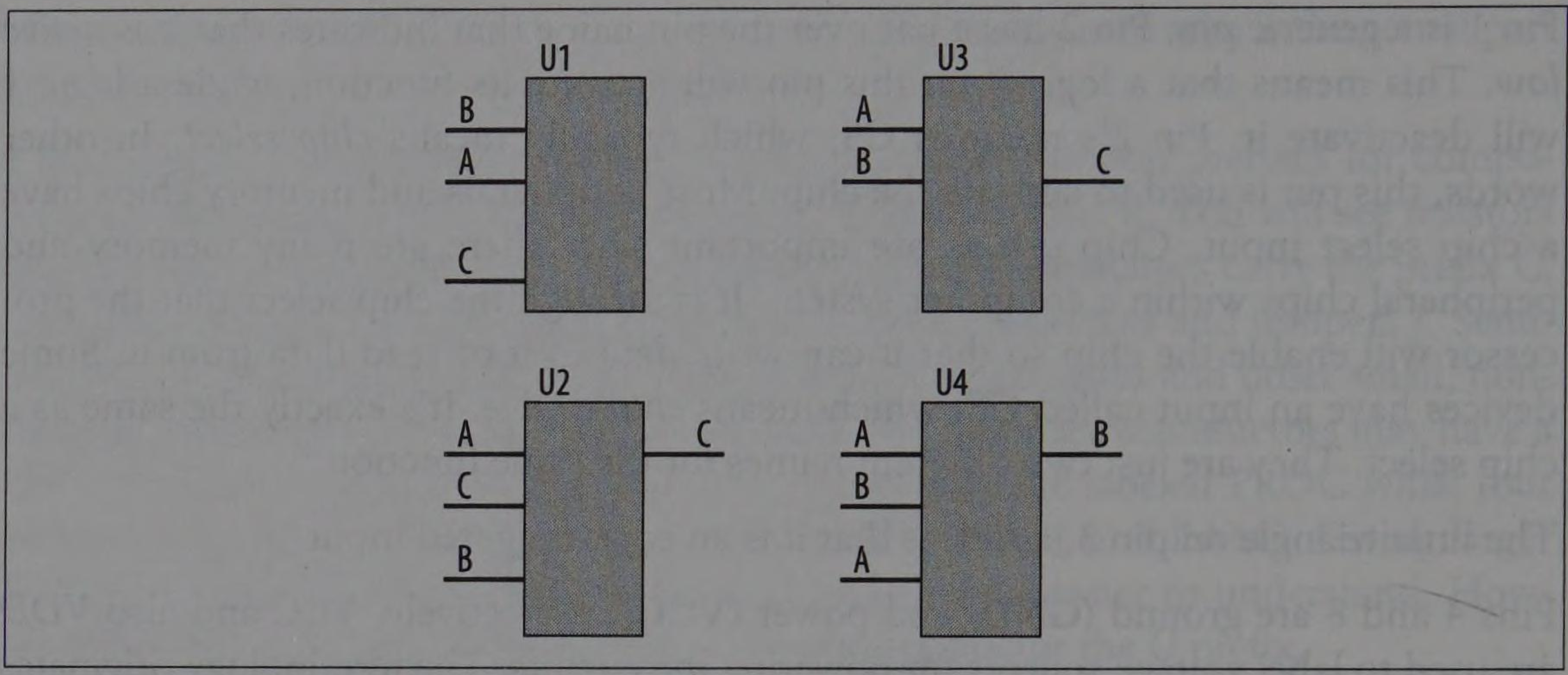


Figure 2-42. Net labels show which pins are connected without the need for drawing every wire

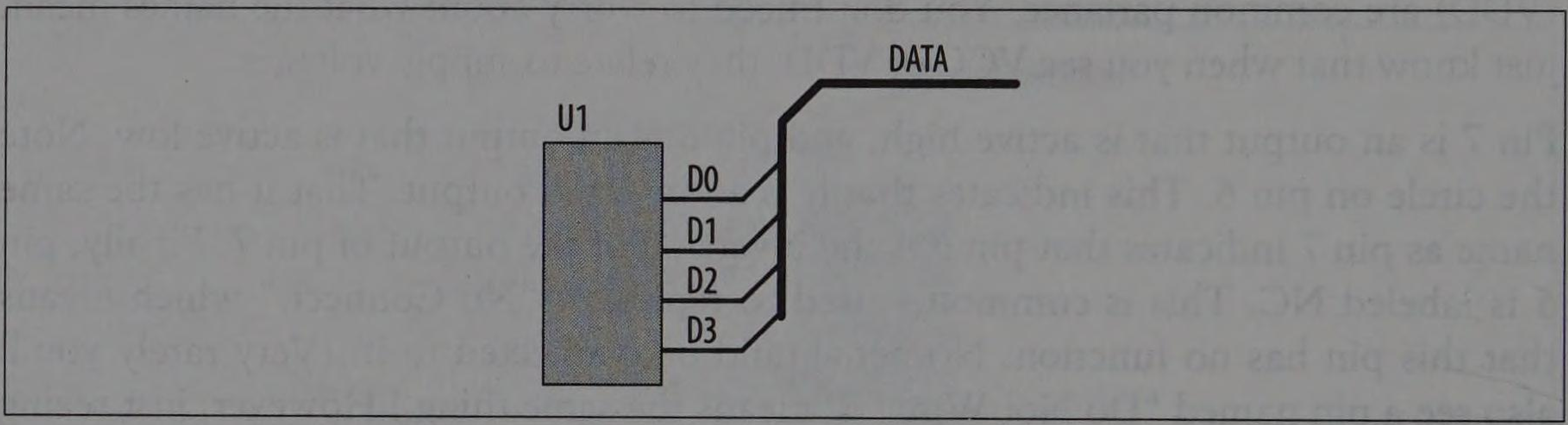


Figure 2-43. Related signals are routed using a bus

Figure 2-44 shows a component with connections to off-sheet objects. In this case, the D0:D3 port is a bidirectional bus, the A0:A3 port is an input bus to this sheet (and therefore an output from another sheet), and the MODE port is an input net to this sheet.

Figure 2-45 shows nets crossing each other. The vertical net on the left *is not* connected to the horizontal net. It simply crosses over on its way to another part of the circuit. The vertical net on the right *is* connected to the horizontal net, and this is indicated by a *junction* dot.

In some hobbyist electronics magazines and old textbooks, you'll sometimes see nets with little bridges as they cross other nets (Figure 2-46). This is definitely *not* the way to draw it—very unprofessional, very uncool.

Figure 2-47 shows common *power ports*. These indicate connections to voltage sources (power supplies) and grounds. The ground symbols all mean a potential of zero volts. The different symbols are used to differentiate between different ground networks. In microprocessor schematics, you'll commonly see the two leftmost ground symbols (usually only one or the other) and rarely see the other two.

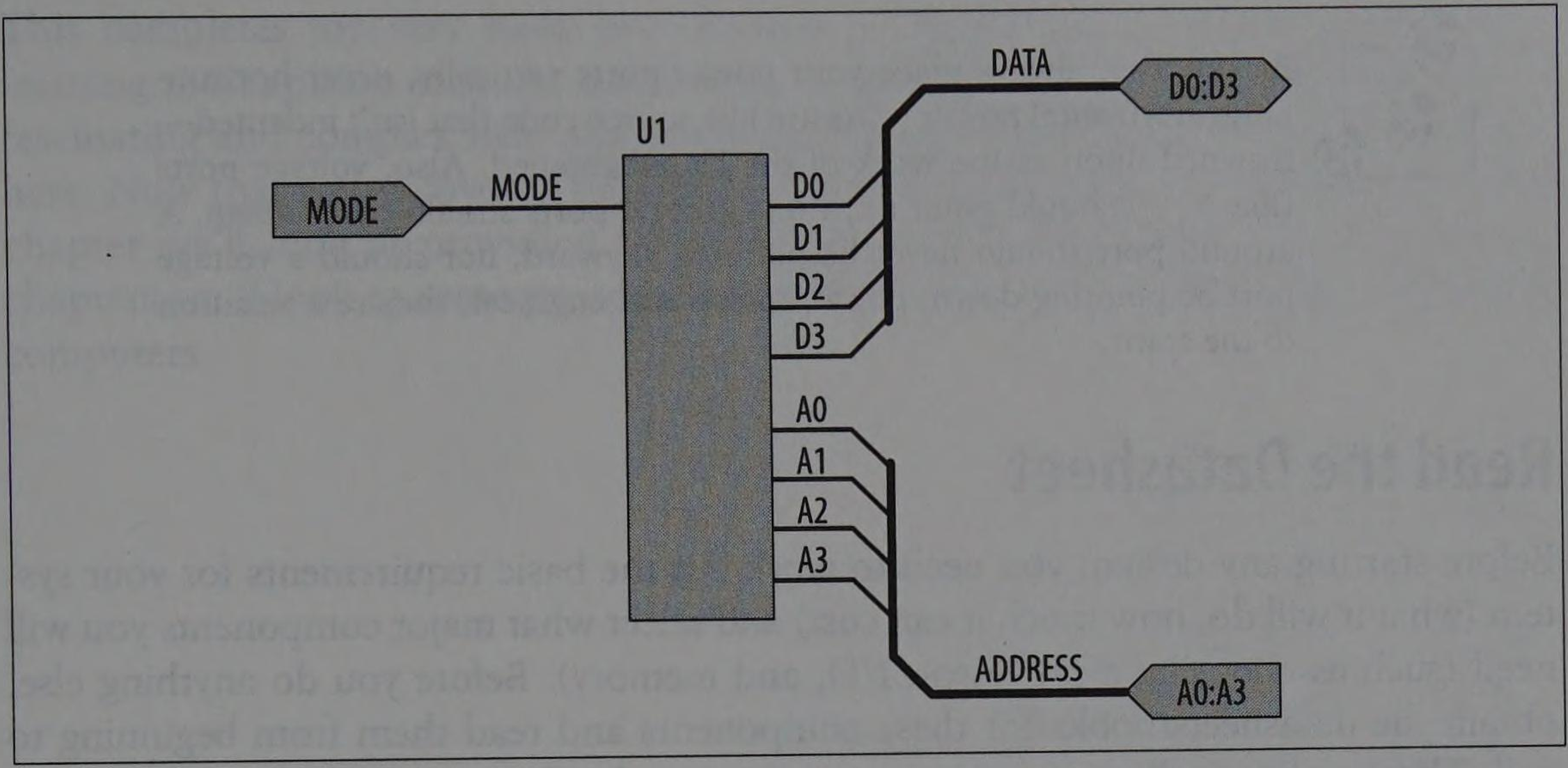


Figure 2-44. Ports indicate that nets are connected across multiple sheets

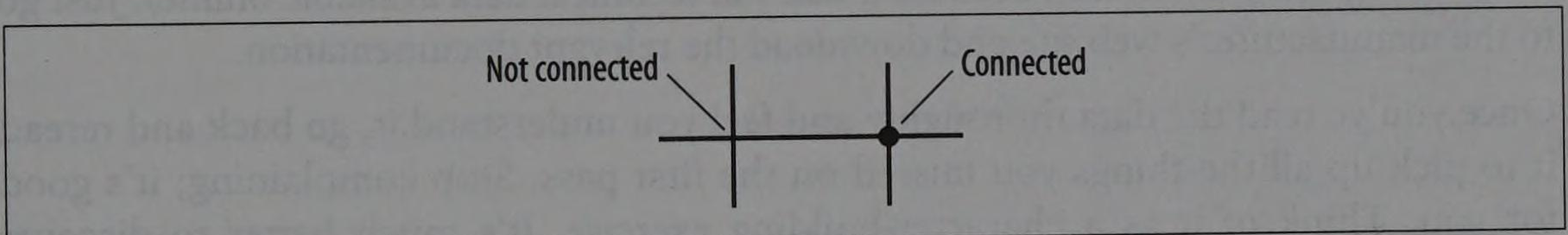


Figure 2-45. Nets crossing

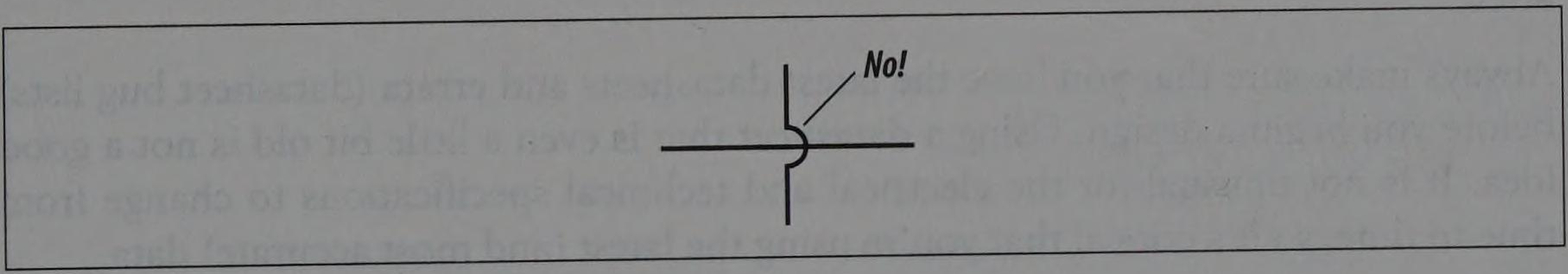


Figure 2-46. How not to draw one net crossing the other

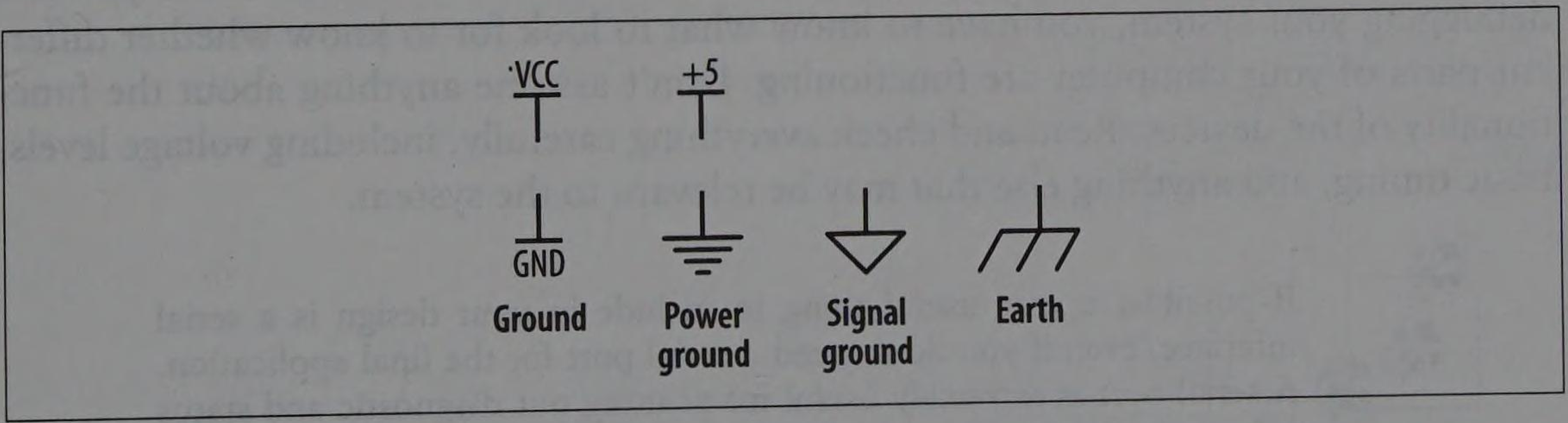


Figure 2-47. Power ports

By the way, always place your power ports vertically, *never* horizontally. Horizontal power ports are like source code that isn't indented—frowned upon as the work of the Unenlightened. Also, voltage ports (like V_{CC}) should point up, while ground ports should point down. A ground port should never be pointing skyward, nor should a voltage port be pointing down. For a professional engineer, they're a vexation to the spirit.

Read the Datasheet

Before starting any design, you need to work out the basic requirements for your system (what it will do, how much it can cost) and select what major components you will need (such as choosing a processor, I/O, and memory). Before you do anything else, obtain the datasheets/books for these components and read them from beginning to end. These can typically be found on manufacturers' web sites (every chip used in this book was specifically chosen because it had full technical data available online). Just go to the manufacturer's web site and download the relevant documentation.

Once you've read the data thoroughly and feel you understand it, go back and reread it to pick up all the things you missed on the first pass. Stop complaining; it's good for you. Think of it as a character-building exercise. It's much better to discover something critical that you've missed before you design and build a computer than after.

Always make sure that you have the latest datasheets and errata (datasheet bug lists) before you begin a design. Using a datasheet that is even a little bit old is not a good idea. It is not unusual for the electrical and technical specifications to change from time to time, so it's critical that you're using the latest (and most accurate) data.

It is very important that you understand how the devices work. When you are debugging your system, you have to know what to look for to know whether different parts of your computer are functioning. Don't assume anything about the functionality of the devices. Read and check everything carefully, including voltage levels, basic timing, and anything else that may be relevant to the system.

If possible, a very useful thing to include in your design is a serial interface, even if you don't need a serial port for the final application. A serial port is *extremely* useful for printing out diagnostic and status information from the system and can be an indispensable diagnostic tool (for both hardware and software). We'll look at serial ports in Chapter 10. The other mandatory debugging tool is the status LED. A flashing LED can tell you volumes about a machine under test if used intelligently by the software and programmer. The more status LEDs you have, the better life will be! (We'll see how to add a status LED in Chapter 6.)

This completes my very basic introduction to electronics. If you're interested in learning more, please seek out some more in-depth books on the topic. It really is a fascinating and complex field and worthy of deeper coverage than I am able to give here. Now that we've covered the basics of what the electrons get up to, in the next chapter we'll look at providing power for your embedded system. In the coming chapters we'll look at some processors and see how to build some simple embedded computers.

CHAPTER 3
Power Sources

The attention span of a computer is only as long as its electrical cord.
—Turnaucka's Law

There is one important aspect that must be included in all embedded computer designs—power. In this chapter, we look at power sources for your computer and voltage regulation to keep your power smooth and reliable.

Your embedded computer system needs electricity. You have several options when it comes to powering your system: coal, nuclear, hydro, geothermal, or batteries. The first four fall under the general category of "juice from the wall."

Juice from the Wall

If your system doesn't need to be portable, this is the most obvious choice. What comes down the pipe is AC and far too high a voltage to be of immediate use to a digital system. It must be converted to a DC voltage of significantly lower magnitude. There are plenty of solutions for doing this. You can use DC lab power supplies, standard PC supplies (probably overkill for your needs), or *AC adapters*. The last of these is probably the best choice for most applications.

AC adapters (also known as *plug packs* or sometimes *power bricks*) are the little black boxes that come with your cell phone and a host of other appliances. They are a cheap, easy, and reliable solution and can be purchased from any good electronics vendor. Typically, they will provide an output voltage somewhere in the range of +5VDC to +12VDC and can supply a current of up to 500mA, depending on the particular plug pack. Choose one that can supply an appropriate voltage and current for your system. One caveat with plug packs is the polarity of the connector. Some plug packs have the positive voltage on the center of the connector jack and ground on the outside. Other plug packs have the exact opposite arrangement! Not knowing which you have could lead to disastrous consequences for your embedded sys-

tem. As always, check the technical data. A better way is to incorporate a bridge rectifier as part of your design (Figure 3-1). The input power is DC, but the polarity of the connection makes no difference. The embedded system uses the output of the rectifier as its power source and has internal voltage regulation. (We'll discuss regulators shortly.)

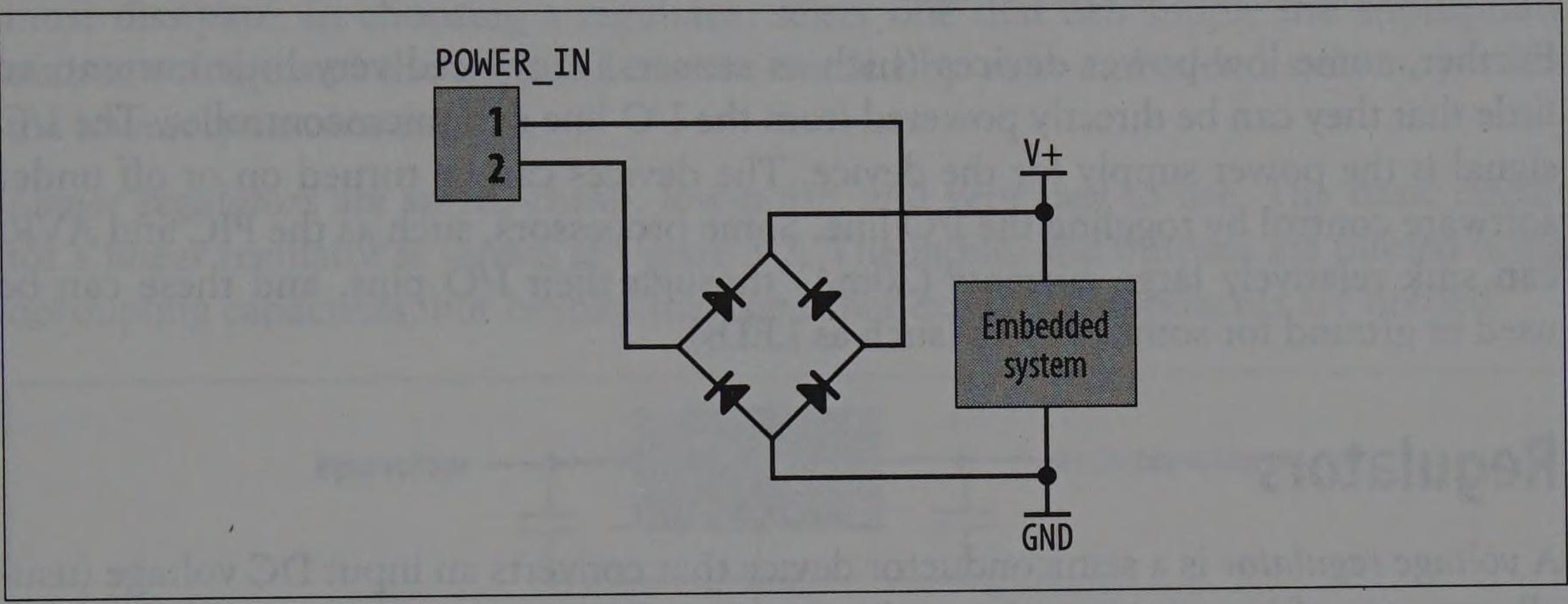


Figure 3-1. A bridge rectifier makes an embedded system "polarity proof"

Batteries

Batteries are easy to use. The only catch is that the battery (or batteries) you choose must supply enough current at the right voltage. With the right choice of battery and a carefully designed system, you can achieve extended operation over very long periods of time. For example, a small PIC- or AVR-based computer can (depending on application and design) operate for up to two years off a single AA battery. A poorly designed system can drain a battery in minutes. A poorly chosen battery unable to supply sufficient current will result in erratic operation or may result in the system being unable to start at all. When choosing a battery, consider not just its average current capability but also its peak current. An embedded computer may need only a constant supply of 20mA but may require as much as 100mA at peak loads. This is especially true of systems using flash memory, which may require high currents during write operations. The battery for such a system must be able to supply not just the continuous load but also the peak load when required.

Power consumption in an embedded system can be reduced in several ways. The use of low-power devices is the most obvious place to start. The power consumption of different devices varies considerably, and many low-power variants of common devices are available. RISC processors often have lower power consumption than comparable CISC processors, so they are often used in preference to CISC in low-power applications. The PIC and AVR microcontrollers can have current draws of less than 5mA (and as low as 10nA when in sleep mode!). This is considerably less than the 35mA of a 68HC11 microcontroller.

Many memory chips and peripherals will enter a low-power mode when they are not in use. However, the power consumption of some devices can be reduced even further. A useful technique for reducing a system's power consumption is to turn off devices when not in use. If the processor is executing code from RAM and outputting data to a serial port, then the power to the ROMs and any other I/O devices may be turned off, as they are not in use.

Further, some low-power devices (such as sensors) may need very little current, so little that they can be directly powered from the I/O line of a microcontroller. The I/O signal *is* the power supply for the device. The devices can be turned on or off under software control by toggling the I/O line. Some processors, such as the PIC and AVR, can sink relatively large currents (20mA) through their I/O pins, and these can be used as ground for some devices (such as LEDs).

Regulators

A *voltage regulator* is a semiconductor device that converts an input DC voltage (usually a range of input voltages) to a fixed-output DC voltage. They are used to provide a constant supply voltage within a system.

While many components in an embedded system can operate from a wide power-supply range, a fixed operating voltage is necessary for such devices as *Analog-Digital Converters* (*ADCs*), since many use the internal power supply as a reference. In other words, the output voltage of a sensor is sampled as a percentage of the voltage supply of the ADC. If the supply is not a known voltage, then any sampling performed by the ADC is meaningless. (We'll look at ADCs in Chapter 12.)

Therefore, a voltage regulator is required to provide a constant voltage source and, thereby, a constant voltage reference. Further, a voltage regulator can assist in removing power-supply noise and can provide a degree of protection and isolation for the embedded system from the external power source. If your system is operating from a battery, the varying current draw of your system can combine with the battery's internal resistance to create a varying supply voltage. The addition of a voltage regulator prevents this from becoming a problem to your embedded system. Including a voltage regulator in your design is good practice. National Semiconductor has a good online tutorial on using and designing voltage regulator circuits. It can be found at *http://www.national.com/appinfo/power/webench*.

The types of regulators we will look at are termed *DC-DC converters*. They take an unregulated DC voltage (often over a range of possible voltages) and provide a constant DC voltage output of a fixed value.

There are three types of DC-DC converters: *linear regulators*, which produce lower voltages than the supply voltage; *switching regulators* that can *step up* (*boost*), *step down* (*buck*), or invert the input voltage; and *charge pumps*, which can also step up,

step down, or invert the supply voltage, but with limited current-drive capability. (Not all charge pumps provide regulated voltage.)

The conversion process of any regulator is not 100% efficient. The regulator itself uses current (known as *quiescent current*), and this is sourced from the input supply. The greater the quiescent current, the more power (and therefore heat) the regulator must dissipate. In choosing a regulator, select one that can supply the appropriate output voltage and the required current needed by your embedded system, yet has the lowest quiescent current.

Linear regulators are small, cheap, low-noise, and very easy to use. The basic circuit for a linear regulator is shown in Figure 3-2. The inputs and outputs are filtered using decoupling capacitors, but beyond that, no other external components are needed.

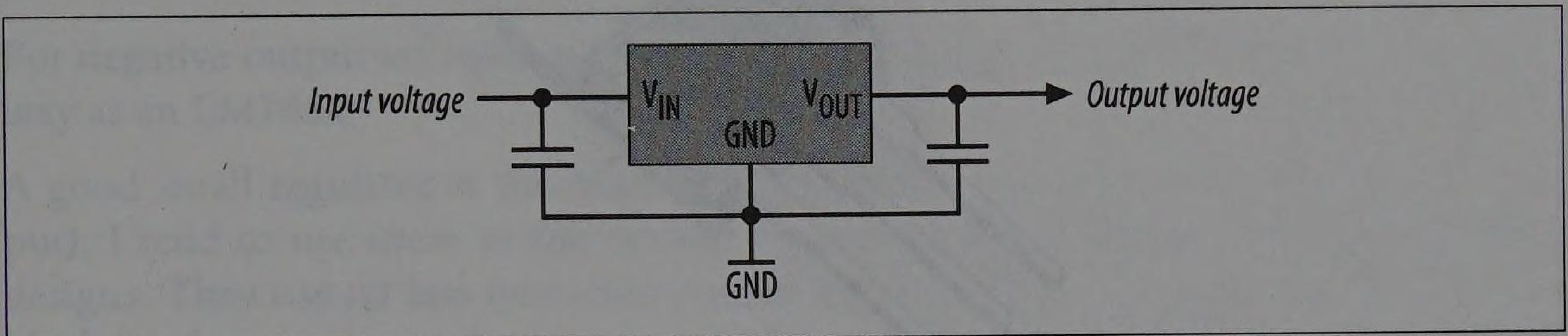


Figure 3-2. Example linear regulator circuit

As well as helping smooth the voltages, the capacitors also help remove momentary glitches in the power source, known as *brownouts*. These momentary drops in power are infrequent, but when they occur they can severely corrupt a computer's operation. Many microprocessors include brownout detectors that will restart the processor if a brownout gets through to the processor's power inputs.

Switching regulators get their name because they switch a power transistor (MOSFET) at their output. They tend to be more efficient than linear regulators in converting the input voltage to the output voltage. In other words, they waste less power during the conversion process. However, their drawbacks are that they require more external components (such as an inductor and diode) and therefore take up more space. They also typically cost more and generate far more noise than linear regulators. Unlike linear regulators, they can step up a voltage as well as stepping one down, and they can also invert. So, for example, a switching regulator can take a supply voltage of 3.6V from a battery and provide you with a regulated 5V supply for your embedded system. Alternatively, a switching regulator may take an unregulated 8V supply and convert this to a regulated -12V. Switching regulators are far more versatile than linear regulators. However, they do require careful design and board layout, so pay careful attention to the directions of the particular component manufacturer. As always, read the datasheets carefully.

Charge pumps, like switching regulators, can step up, step down, or invert voltages. Unlike switching regulators, they require no external inductor. However, due to their limited capacity to supply current, they are not commonly used. The MAX3222 (and

similar devices) discussed in the chapter on serial interfaces, use internal charge pumps to generate the +12V and -12V required for RS-232C-level shifting.

Literally thousands of voltage regulators are available. Probably the most commonly used are the LM78xx linear regulator series, made by several manufacturers such as Fairchild (*http://www.fairchildsemi.com*), Semelab (*http://www.semelab.co.uk*), and ST Microelectronics (*http://www.st.com*). They typically come in a TO-220 package (Figure 3-3) and have a metallic attachment point for a heat sink. The regulator is normally mounted flat against the circuit board, and the pins are bent 90 degrees downward.

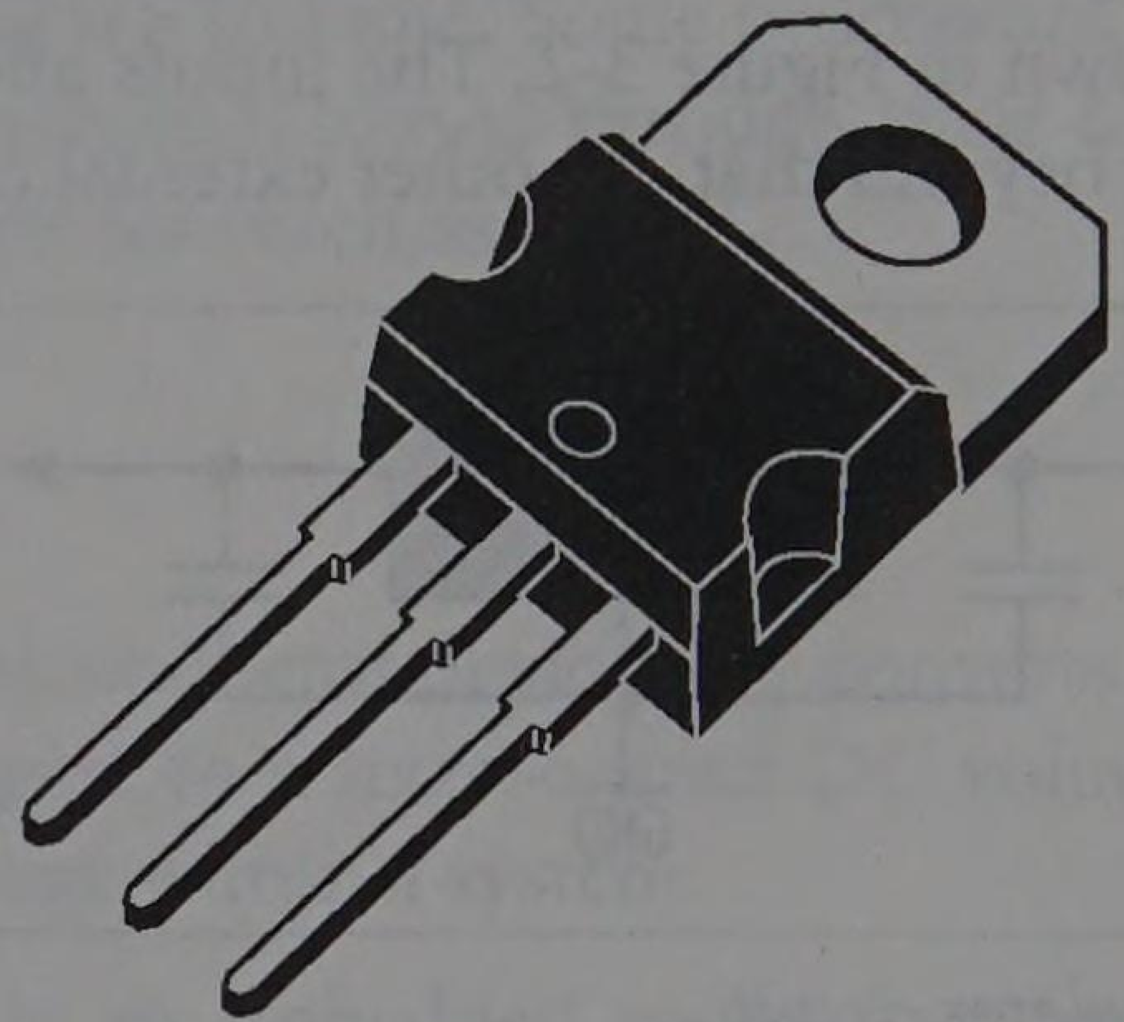


Figure 3-3. LM78xx

The part number designates the output voltage. For example, an LM7805 provides a 5V regulated output, while an LM7812 gives a regulated 12V output. They can provide an output current of up to 1 Amp (and as much as 2.2 Amps peak) with a quiescent current of between 5mA and 8mA. They also feature overload and short-circuit protection. Table 3-1 lists the regulators, their input voltage ranges, and their output voltages.

Table 3-1. LM78xx voltage regulators

Part	Output (V)	Input range (V)
LM7805	5	7–25
LM7806	6	8–25
LM7808	8	10.5–25
LM7809	9	11.5–25
LM7810	10	12.5–25
LM7812	12	14.5–30
LM7815	15	17.5–30
LM7818	18	21–33
LM7824	24	27–38

The LM78xx is simple to use. Decoupling capacitors (nominally between 10μF and 47μF) are required on the input (pin 1) and output (pin 3), as shown in Figure 3-4. Pin 2 is connected to ground.

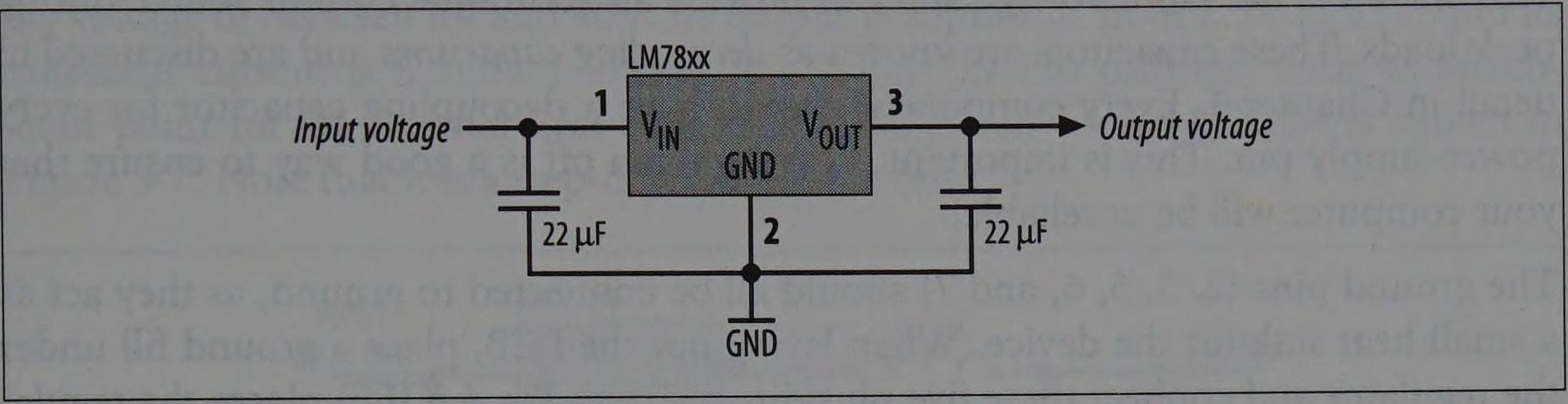


Figure 3-4. LM78xx circuit

For negative output voltages, use an LM79xx regulator. It's used in exactly the same way as an LM78xx.

A good small regulator is the Maxim MAX603 (5V output) or MAX604 (3.3V output). I tend to use these as the default workhorse regulators for many of my small designs. They use far less quiescent current than LM78xx regulators and as such are ideal for low-power or battery-operated systems. These switching regulators are available in tiny surface-mount SO-8 or in standard DIP (Dual Inline Package) packages (discussed in Chapter 4) and require only two external components. The astute of you may ask how it can be that a switching regulator needs only decoupling capacitors. Where are the inductor and diode? The answer is that Maxim has put everything on the one chip, making life much easier.

The MAX603/604 can provide up to 500mA of current and can operate from an input voltage of between +2.7V and +11.5V DC. They have built-in protection in case you inadvertently switch power and ground, and they consume as little as 15μA of current. As such, they are ideal for use in low-power, embedded computers. The schematic for using a voltage regulator such as a MAX603 or a MAX604 is shown in Figure 3-5. In this case, the input supply is a battery, but it could just as easily be a DC plug pack or some other sort of supply.

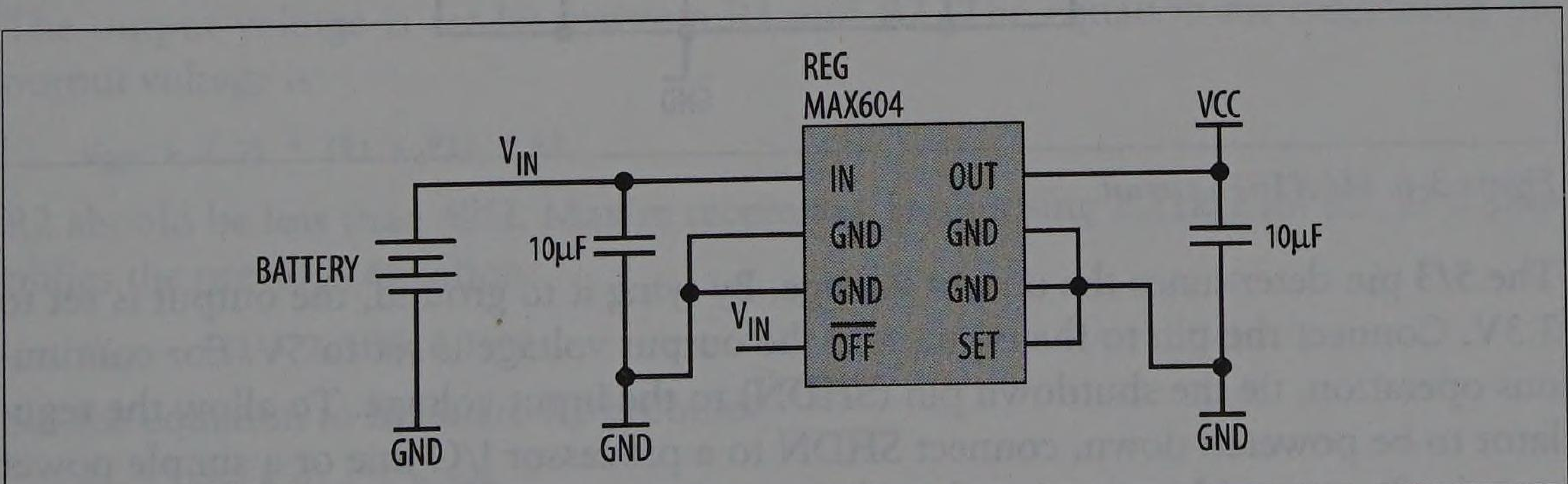


Figure 3-5. MAX603/MAX604 circuit

A capacitor is required at the output (pin 8 of the regulator) to filter the voltage. This capacitor forms part of the regulation circuit, and the device will not regulate without it. The second support component required is a capacitor on the input (pin 1). This stabilizes the input voltage and can provide an additional current source during peak loads. These capacitors are known as *decoupling capacitors* and are discussed in detail in Chapter 4. Every component should have a decoupling capacitor for every power-supply pin. This is important. Leaving them off is a good way to ensure that your computer will be unreliable.

The ground pins (2, 3, 5, 6, and 7) should all be connected to ground, as they act as a small heat sink for the device. When laying out the PCB, place a ground fill under the regulator and connect these five pins directly to it. Pin 4 (**OFF**) places the regulator in shutdown mode. For constant operation, this pin is connected directly to the power source, so that the device is always on.

A MAX604 gives a regulated 3.3V output. For a 5V output, replace the MAX604 with a MAX603. The circuit is otherwise the same.

The MAX1615 is a linear regulator that can operate from a supply range of between 4V to 28V. It is tiny and is capable of supplying 30 mA of output current at either 3.3V or 5V. Now, 30mA is not much current, but for very small (battery-powered) applications it may be sufficient. The MAX1615 has a shutdown input, allowing an external system to power it down. One use for this regulator could be as a power source to subsystems within an embedded computer, allowing the host processor to turn them off when not in use. Before turning off a subsystem, ensure that its "absence" won't adversely affect the functionality of the rest of the system.

The basic schematic for a MAX1615 circuit is shown in Figure 3-6.

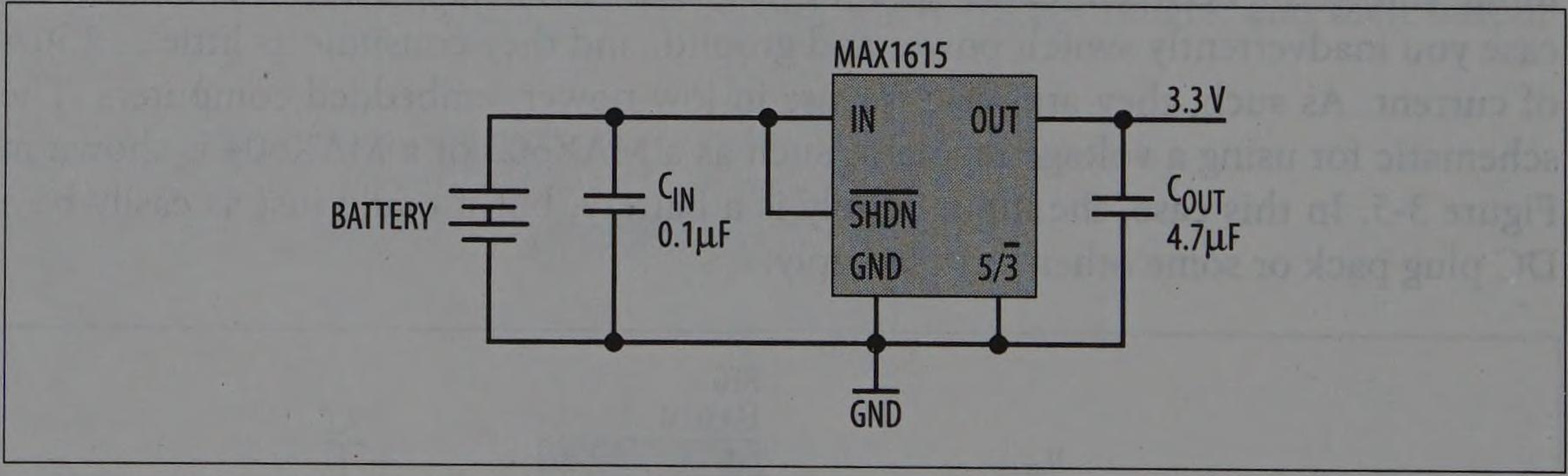


Figure 3-6. MAX1615 circuit

The **5/3** pin determines the output voltage. By tying it to ground, the output is set to 3.3V. Connect the pin to the input, and the output voltage is set to 5V. For continuous operation, tie the shutdown pin (**SHDN**) to the input voltage. To allow the regulator to be powered down, connect **SHDN** to a processor I/O line or a simple power switch. (It goes without saying that if you're driving **SHDN** with an I/O line of the

processor, then that processor must have a power source other than this regulator! Otherwise, it could lead to some interesting situations.)

For higher-current situations, the MAX724 can supply up to 5A from an input supply voltage of between 8V and 40V. Its output is adjustable from 2.5V to 35V, and its quiescent current is 8.5mA. It comes in a 5-pin TO-220 package, with an attachment point for an external heat sink. The basic circuit for a MAX724 is shown in Figure 3-7. Note that it is a step-down regulator only.

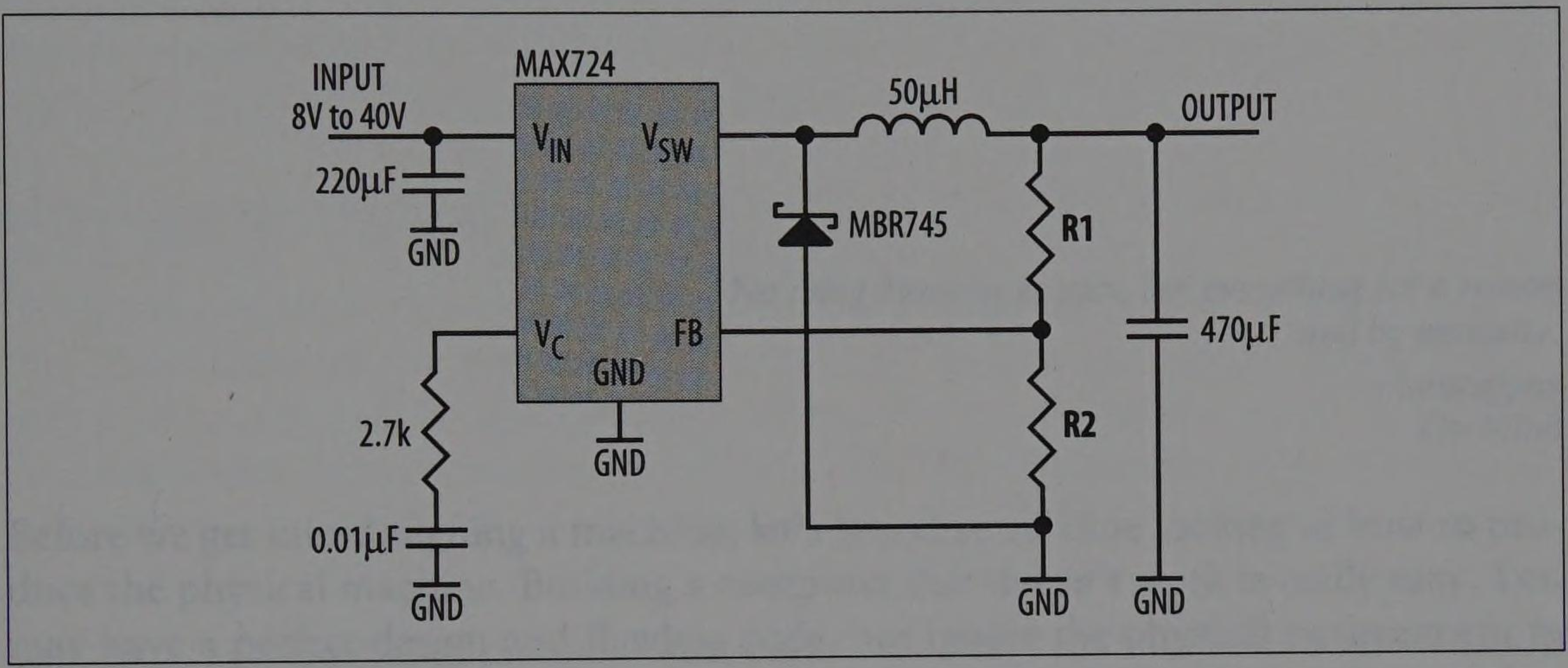


Figure 3-7. MAX724 circuit

The inductor is nominally 50µH but can be any value in the range 5µH to 200µH. When the output pin V_{SW} turns off, the diode provides a path to ground for the inductor current. The inductor chosen should have a high saturation current. If it doesn't, the effects will be disastrous. The Maxim datasheet provides detailed information on selecting an appropriate inductor.

Maxim recommends that a Schottky diode, such as an MBR745, be used due to the fast switching times required. Both the regulator's input and output require large decoupling capacitors to filter out ripple. The capacitors must have low *ESR* (Equivalent Series Resistance) over the expected temperature range and operating lifetime.

The output voltage is set by resistors R1 and R2. The equation for calculating the output voltage is:

```
VOUT = 2.21 * (R1 + R2) / R2
```

R2 should be less than 4kΩ. Maxim recommends choosing 2.21kΩ for R2, as it simplifies the previous equation:

```
VOUT = (R1 + 2.21k) / 1000
```

So, the equation to calculate R1 becomes:

```
R1 = 1000 * VOUT - 2.21k
```

The regulators I've covered so far should be useful in most situations. If you're designing a machine with special requirements, there is bound to be a regulator that will suit your needs. Just spend some time browsing component manufacturers' web sites and see what they have to offer.

CHAPTER 4

Building It

No thing happens in vain, but everything for a reason and by necessity.

—Leucippus
On Mind

Before we get into designing a machine, let's spend some time looking at how to produce the physical machine. Building a computer that doesn't work is really easy. You may have a perfect design and flawless code, but ignore the physical environment in which the machine exists, and you'll have built yourself a very intricate paperweight.

In this chapter, I'll also show you how to lay out a circuit board (and where to be especially careful) and how to debug your hardware. In particular, I'll examine how to physically produce the design for the ATtiny15 computer, which is presented in Chapter 6. I assume that you're hand building in small quantities, so I'll target the discussion accordingly. What I present here is not the state of the art in circuit board design or assembly, but guidelines for cottage-industry computer production. If you need to make production runs of hundreds of thousands, either you already know what you're doing (and can skip this section) or you need to talk to a professional.

Avoid Noise

Digital systems are inherently analog in operation. Digital signals suffer degradation and noise due to analog effects present in the system. Spurious noise or reflections from nearby electrical machinery or radio transmissions can induce signals within your circuit that can cause false events to occur or even prevent a digital system from functioning at all. The one way to ensure that your system is immune to electromagnetic interference is to avoid the use of electricity! Unfortunately, the steam-powered microprocessor is not a reality, so if your system is to operate reliably in the real world, you must take electromagnetic effects into account. What follows is not a comprehensive overview of noise and associated problems and solutions. It is far too complex a field to cover properly here. What I will do is just provide an introduction.

The subject is worth seeking out more detailed information and understanding these concepts more thoroughly.

> The United States, Australia, the nations of the European Union, and many other countries have very strict guidelines and requirements for electromagnetic emission and immunity. The recommendations presented here are good practice only and are not necessarily sufficient to warrant compliance in your country. Therefore, it is important that you check your local regulations and ensure that you meet the appropriate requirements. Compliance cannot be guaranteed just by design. A system must be tested to ensure that it is compliant.

Noise can be a significant problem in digital systems. Noise can disturb signal transmission leading to corrupted data or may even cause a program to crash. Problems with an embedded system may or may not be noise related. Inadequate power-supply levels, insufficient decoupling capacitors, marginal timing tolerances, and software bugs can all cause seemingly random glitches in operation. However, even a well-conceived system can be disrupted by noise. There may be noise problems inherent in the design. Switching noise from integrated circuits, ringing, and cross talk are all due to aspects of the designed system. Other forms of noise may be due to environmental effects (such as nearby motors or radio emissions). The bad news is that electromagnetic problems are getting worse for digital systems. The environment is bathed in ever-increasing emissions from radio, TV, and cell phone towers. At the same time, integrated circuits are becoming increasingly sensitive as designs move toward higher speed and lower power operation.

These sources of noise may not be present during the design and test phase of the system but will only manifest themselves once the system is out in the field and in service. A crashed embedded system may be due to a hardware problem, a software problem, or the fact that the factory a block away turned on a compressor, which caused a spike on the power supply. Field problems created by noise may occur only very occasionally and can often be very difficult to track down. It is not unusual for some problems to occur only once every few days. Any problem is unsatisfactory and must be fixed, but identifying the cause is not always easy. It is better to design the system from the beginning to be as immune to these problems as possible. You have to consider not just what emissions your system may produce, but also how it may be susceptible to external effects. This will not guarantee your system will be problem-free, but every bit of immunity helps.

Electromagnetic interference (*EMI*) is noise generated by sources external to the embedded system. Some examples of EMI are motors, switches in power consumption, fluorescent lighting, RF emissions, and electrostatic discharges. All can be significant sources of noise. For example, turning a machine with an electric motor on or off can cause a 1000V spike on the AC power-supply line. An *electrostatic discharge* (*ESD*) can send a spike of 35kV from a finger into an integrated circuit with a

current rise time of 4 Amps per second! This can be enough to permanently damage a sensitive chip. Cars are particularly noisy environments. The 12V supply line to the automotive electronics may be reversed, driven at voltages ranging from 6V to as much as 24V, and have 400V transients spikes. All of these can have very adverse effects on the operation of an embedded system.

In any circuit, there is a wire carrying current in and a wire carrying current out. Current flowing through a wire generates a magnetic field around that wire. Such a magnetic field can be a source of EMI. The intensity of the magnetic field felt is inversely proportional to the distance from the source of the field. The orientation of the field relates directly to the direction of current flow in the wire.

Minimize the Current Loop Area

Current flows through a system via the power and signal connections and back to the power supply (thereby completing the *circuit*) through ground. Ground thus forms the *return path* for current flowing within the system. If the signal wire and return wire are located close together, the magnetic fields generated by the currents in the wires cancel out within a short distance of the wires. This is known as *minimizing the current loop area*. The objective is to keep all signal and return paths as close together and as short as possible. Where there are many current loops present in a system (as is common in many large, high-speed, digital systems), a *ground plane* is used to minimize loop area. A ground plane is a large conducting surface that can serve as the current return path for all loops in the circuit. A ground plane is often implemented as a complete, internal PCB layer.

Capacitive coupling is the coupling of electric fields. A signal on one wire, through its associated electric field, can capacitively induce a phantom "signal" in an adjacent signal line. This is known as *crosstalk* in digital systems. If not designed correctly, the magnitude of the crosstalk in a system can be significant and can easily cause a crash.

Capacitive coupling may be reduced by shielding the signal lines with an electrostatic or *Faraday* shield. This shield is a metal conductor placed between the capacitively coupled elements. The shield is simply part of the PCB (discussed later in this chapter), formed in the same way as the circuit tracks, and is usually grounded (though not always). The shield may be a simple ground plane under an integrated circuit to protect it from signal lines on the underside of the circuit board. Signal lines may be shielded from each other (if necessary) by placing a ground line between them.

Keep the Power Smooth

The principle of keeping current loops small applies as much to power lines within a system as it does to signal lines. However, keeping the loop area small for power is difficult. Power must be distributed throughout the circuit, and to effectively route this throughout a *printed-circuit board (PCB)*, and keep the loop area small, is very

difficult. The power lines can therefore be susceptible to noise, and this can cause major problems to the circuit.

The solution is to provide a path to ground for any noise present in the power supply. This may be done locally for each component in the circuit. It is achieved by adding a *decoupling capacitor* between power and ground for each integrated circuit. The capacitor decouples the noise from the power source and provides a path to ground for it. In this way, noise is removed from the power supply, and the chips have a constant and clean voltage source. The decoupling capacitors should be placed as close as possible to the power pins of the devices. Surface-mount capacitors have very low inductance connections and so are preferable. Ceramic capacitors are normally used for decoupling capacitors due to their low resistance.

The capacitor has the added advantage of acting as a current source for the device when the device must switch its outputs or internal state. As such, it represents a current source with a much smaller loop area. Generally, the circuit board will be decoupled by a large (22–100μF, say) electrolytic or tantalum capacitor placed near the power input, and each integrated circuit will be separately decoupled by 10nF ceramic capacitors. Multiple decoupling capacitors, one for each power pin, improve the situation. You need to ensure that all frequencies that may affect the circuit have a low impedance path to ground. To this end, several capacitors (100nF, 10nF, and 100pF) can be used to decouple a wide range of frequencies and thereby remove noise from the power-supply circuits.

Additionally, an onboard voltage regulator can provide a degree of isolation between your circuit and the external power supply. You can never have enough decoupling capacitors (within reason).

The Importance of Decoupling Capacitors

Many years ago, when I taught at a university, I had a final-year student undertaking a project to design and build a 64-bit workstation using the now long-vanished Motorola 88110 MIMD processor. The machine failed to work, because she had neglected decoupling capacitors in the design. When the fault was pointed out and corrected, the machine roared into life. Those simple capacitors made the difference between a working 64-bit computer and one that never managed to climb out of reset.

Noise can also be present in ground lines. Ground is not always at the same potential in all locations. There can be a voltage difference between the local grounds in different parts of a circuit that are both connected to "ground." This voltage difference can drive currents of several Amps through the ground line. This is referred to as a *ground loop* and can result in serious problems. Shielding, decoupling, and minimizing the current loop area can help protect against ground noise.

Ground bounce is ringing (oscillation) on signal lines caused when one or more outputs on the same device are being switched from high to low. This ringing can be of significant amplitude and can adversely affect the system. Some devices are designed so that ground bounce effects are minimized. Using devices in packages with shorter leads (such as PLCC and surface-mount) can reduce ringing effects. Devices with ground and power pins toward the center of the package have shorter lead lengths, and this also helps to reduce ringing effects. Termination techniques can also help to reduce ground bounce.

Another effect you are likely to encounter is caused by the simultaneous switching of several outputs at once. Like ground bounce, it is due to parasitic effects relating to packaging and internal wiring of the chips. When several outputs switch at once, these effects can cause a delay of several nanoseconds in the changing output signals. Component datasheets usually specify timing parameters for a single changing output. You need to consider the effect that several changing outputs may have on your circuit.

How to Destroy a Computer Without Really Trying

If you walk across a carpet on a dry day or rub a cat against a plastic surface, a static charge will build up. If you or the cat then touches something metallic, the jolt you both feel is an *electrostatic discharge* (*ESD*). An ESD can destroy an integrated circuit permanently. The ESD may be too small to be felt, but it can still send a semiconductor to that great beach in the sky.

Many integrated circuits have internal protection against ESD. This protection is sufficient to safeguard the device against the charge buildup that can occur during normal handling. It is not, however, sufficient to protect the device against the huge electrostatic sparks that can sometimes occur. Once a device is in-circuit, it should not be considered safe. It is possible for a processor to be destroyed by a spark received when a typist puts fingers on a keyboard. The spark, like a lightning strike, will attempt to find a path to ground, even if that means traveling down a data line and through the processor to get there.

One solution is to include a buffer chip to isolate the important components in the system from those that may come in contact with an ESD. This is not an ideal solution. It simply means that the buffer will be destroyed instead of the processor. You still have a system that has failed.

Transient suppressors that can provide protection are available. Such suppressors act as an open circuit at normal voltage levels but conduct power to ground at higher voltages. At no time should a signal ground be used to earth an ESD. Most integrated circuits don't respond very well to having their ground pins raised to several hundred volts above their power supply by an ESD!

Many semiconductor manufacturers, such as Maxim, are now building protection into their devices that can withstand ESDs of 15kV or more.

When handling chips, it's a good idea to use a *grounding mat*. This is a conductive sheet that you connect to the ground of a handy power supply. You then wear a *grounding strap* around your arm, connecting you to the grounding mat. Thus, any electrostatic charge is dissipated and never gets the chance to build up.

> If you're using a grounding mat, don't forget to take your embedded system off it before powering up. The ground mat is conductive, and powering up a system while it is in contact can be disastrous!

There are several ways of fabricating a computer (or any other circuit). Let's take a look at them.

Quick-and-Dirty Construction

It is possible to build very simple circuits by just soldering the components together in free space. For example, with the AVR design in this chapter, the leads of the watch crystal can be soldered directly onto the pins of the processor, with the crystal lying across the top of the processor. Wires are soldered onto the pins bringing in ground and power and connecting the processor's I/O to the outside world. This technique is variously referred to as "a rat's nest," a "bird's nest," or "what the hell is that?"

This is a quick-and-dirty method, useful for rapid prototyping of extremely simple circuits. It's not really recommended, but you can get away with it in a pinch. Don't try it with anything that is even slightly complicated or running at any reasonable speed. If you do, you'll spend more time debugging the construction than debugging the actual design or code!

Breadboarding

Breadboards are plastic blocks with arrays of holes. They are designed to hold DIP-packaged integrated circuits and discrete components. The term "breadboard" dates back to the olden days when valve radios were constructed on a base of solid wood (a cutting board for bread). The term has stuck, and the modern breadboard can still be found in electronics hobbyist stores and even the occasional university teaching lab.

While it is possible to build very low-speed microprocessor systems and general digital circuits on breadboards, try not to. There be dragons! As a general rule, breadboards are *bad news* and you should avoid using them at all costs. (Think of them as the hardware equivalent of COBOL.) Breadboards suffer from excessive capacitance, crosstalk, and noise susceptibility, which makes them completely inappropriate for microprocessor system construction. They can also suffer from mechanical failure (leading to short circuits) after extended use. Circuit interconnections on a breadboard are done with small sections of wire, which make great little antennas. They will pick up every scrap of stray electromagnetic radiation and channel it

straight into your circuit! This is not the way to construct a robust and reliable system. If you really must, you could probably build the ATtiny15 or PIC12C805 computer on a breadboard, using their internal RC oscillators. But I'd advise against using breadboards for anything that uses a crystal or that has any fast-switching digital signals.

Wirewrapping

Once common as a construction technique, *wirewrapping* is now quite rare. It is intended for use with DIP-packaged integrated circuits, which are mounted on sockets with long pins (0.6"). Special tools (known as *wrapping tools*) allow you to quickly and efficiently wind thin wire around the pins. The pins are square in cross-section, and wrapping a wire around a pin forms a *cold weld*, a tight electrical connection with no soldering. Thus, a circuit is constructed by individually wiring point-to-point each connection within the system.

Wirewrapping is a very fast prototyping technique and is very robust and reliable. In the early days, NASA used to use wirewrapping for constructing spacecraft avionics, and many mainframe computers were built using the technique. Wirewrapping is good for prototyping (especially if you're unclear as to the final form of the design and expect to make lots of changes to the hardware) or for building one-off designs. If you intend to make more than one computer based on your design (and you probably will), then skip wirewrapping and do it on a printed-circuit board.

Printed-Circuit Boards

Printed-circuit boards are epoxy-bonded fiberglass sheets, plated with copper. The copperplating is etched away, leaving tracks (traces) that form the interconnections of the circuit. PCBs are very reliable and are the only option if you intend to produce more than one system. It is possible to etch your own PCBs, but commercial PCB production isn't that expensive, and it is worth the cost to get professionally produced boards.

EDA (*Electronic Design Automation*) software is used to create the schematic and PCB design. The most popular EDA software comes from Mentor Graphics (*http://www.mentor.com*) and Protel (*http://www.protel.com*). There is also a GNU (*http://www.gnu.org*) PCB editor (called PCB) that is freely available. Such programs normally come with several tools, allowing schematic entry, netlist generation (a list of what needs to be connected to what), PCB layout, manual routing (making the connections), and autorouting. There's a great temptation to use autorouters, as they simplify the process of generating the PCB by getting your workstation to do the hard work of routing. However, I prefer to lay out the circuit board myself. (I've seen some autorouters make a real pig's breakfast of a design.) Routing the board manually can take a long while, but it is often worth the extra effort. It can also be very absorbing, much like spending hours in deep meditation. (It's very Zen.)

PCBs can be single-sided (one layer), double-sided (two layers), or 4-layered, 6-layered, 8-layered, 12-layered, or more. The more layers you have, the easier it is to route your interconnections, but the costs of fabrication go up considerably with extra layers. Further, it's much easier to debug a 2-layered board than a 12-layered board. With additional layers dedicated to power and ground planes, your system will have greater noise immunity. While not so critical for slow 8-bit systems, they are mandatory for high-speed computers.

Multilayered boards will be plated through, meaning that there will be metallic connections through the holes in the board, connecting traces of different layers together, as appropriate. A *solder mask* is the (normally) green coating on circuit boards and prevents solder flowing between pads and tracks during construction. It is possible to order commercial PCBs without plating-through and without solder mask, but the small amount you will save is not worth the hassle.

The *overlay layers* (also known as *silkscreen layers*) are painted on and contain labels (such as R30 or RAM4) showing component placement, used during construction. The overlay layers are optional. If the boards are to be manually populated with components by someone else, the overlay layers are helpful during construction. If you're building them yourself, then you can easily do without the overlays and save a few bucks.

> A trick if you're skipping the overlays is to place component information as text on the copper layers. Just be sure to avoid making contact with the circuit tracks!

The external copper layers are called *top* and *bottom* (no surprises there). Traditionally, the top layer was called the *component layer*, and the bottom layer was called the *solder layer*, since components used to be mounted on top, and their pins soldered underneath. However, most modern circuit boards place components on both sides and are soldered on both sides. Thus, the terms "component layer" and "solder layer" are seeing less use.

There are also internal copper layers for multilayer boards, *mechanical layers* (indicating any special physical features), the *keepout layer* (showing the actual PCB shape), and others. In four-layer boards, it is common practice to use the outer layers for signals and the internal layers for power and ground. This not only provides shielding, it also minimizes the current loop area, thereby giving your design greater stability.

The five types of objects that can be placed on a copper layer are tracks, individual pads, components (arrays of pads grouped together), vias, and fills.

Tracks are used to interconnect components. Track width is expressed in thousands of an inch (mils) or in millimeters (mm). Tracks can be of varying thickness, and often a PCB will have different widths for different tracks. The fatter the track, the more current it can carry. The thinner the track, the easier it is to fit more tracks in a

given space, therefore, the easier it is to route the PCB. Table 4-1 gives a general guide to the current-carrying capacity of different track widths (1 oz. copper), for a temperature rise of +10 degrees C.

Table 4-1. Track width versus current flow

Mils	mm	Amps
8	0.2	0.5
12	0.3	0.75
20	0.5	1.25
50	1.25	2.5
100	2.5	4
200	5	7
325	8.12	10

Check with the company doing your PCB fabrication as to what tolerances they can manufacture to. There's no point in doing a PCB with 4 mil tracks if your local PCB fab company can only go as small as 8 mils.

Pads are used to mount component pins, and they can be either round, rectangular, or oval. They consist of a hole and a copper surround. A pad for a component in a DIP, for example, will be a multilayered pad, meaning that the pad appears on all copper layers, and the hole is drilled through the entire PCB. A surface-mount component will have pads that appear on one layer only (Figure 4-1). An array of pads grouped together to form a component package is known as a *footprint*. Surface-mount components have holes of zero diameter (in other words, they aren't drilled). Surface-mount components are small with "gull-wing" pins that mount flat on the PCB. They are less susceptible to noise interference than the older DIP style of packaging. However, DIP (through-hole) components may be easily mounted in sockets and are therefore easily removed during debugging. DIPs are sometimes preferable (although not always feasible) during early development, while surface-mount is the only option for production.

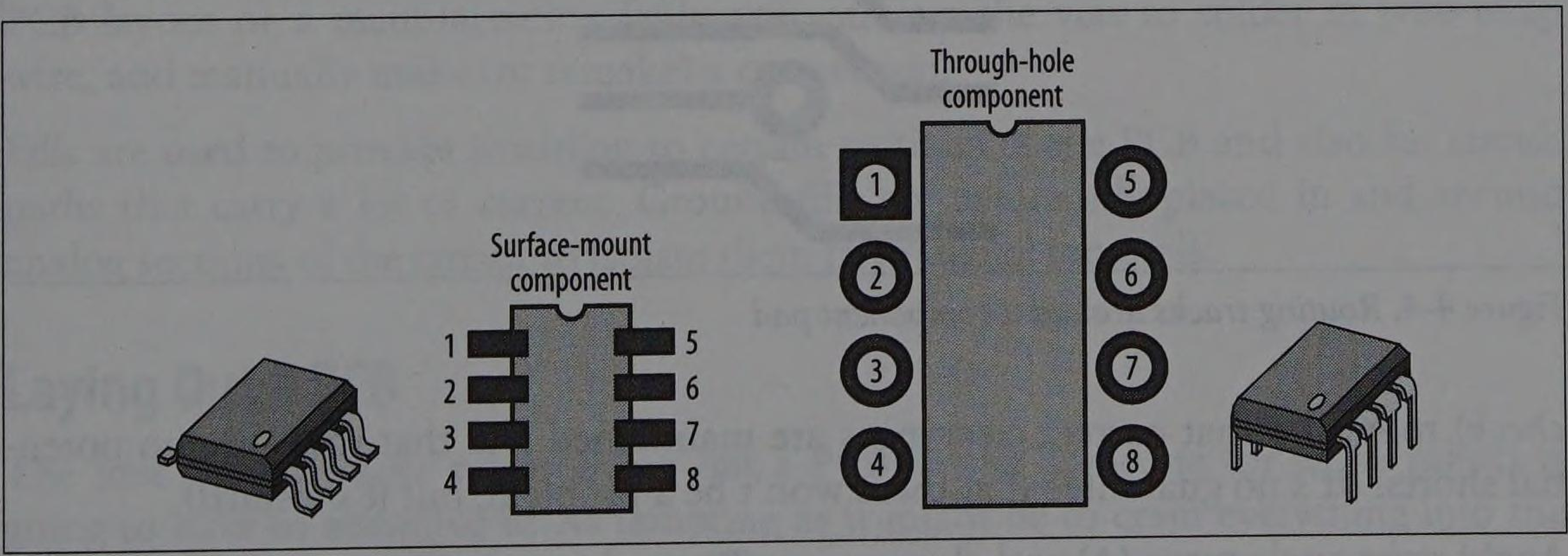


Figure 4-1. Footprints of surface-mount and through-hole (multilayer) components

Tracks entering a pad should aim directly for the pad center, as shown in Figure 4-2 and *not* as in Figure 4-3.

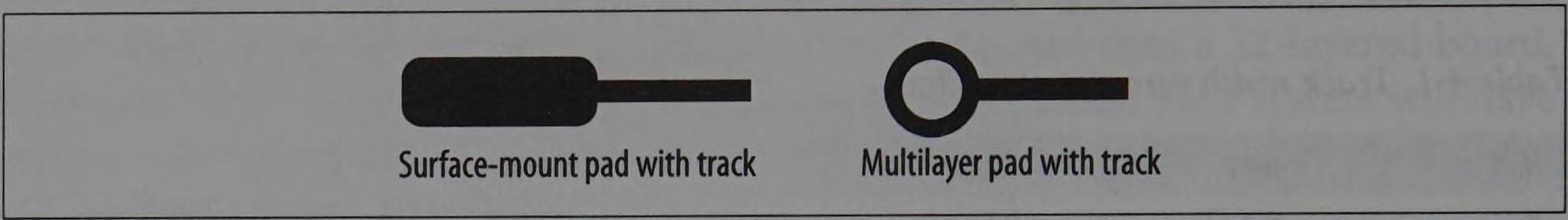


Figure 4-2. Surface-mount and through-hole pads

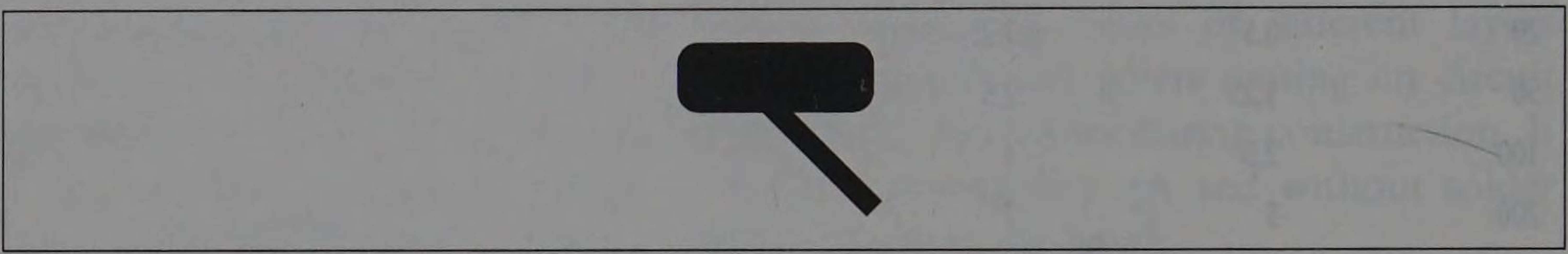
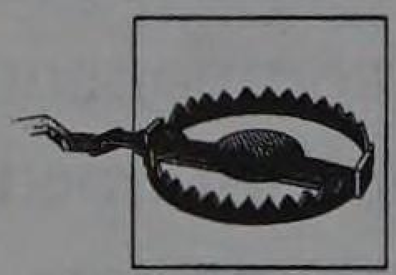


Figure 4-3. The incorrect way for a track to enter a surface-mount pad

When specifying the pads for a component, ensure that the pad size is large enough to accommodate the pins and to allow enough space onto which to solder. Also, ensure that the holes (for through-hole components) are large enough to take the pins. A standard DIP pin will happily go into a 0.7mm hole, while a DB connector requires 0.9mm holes for the signal pins and 3mm holes for the mounting pins.



Don't assume that the libraries that came with your PCB CAD package have the pads, spacings, or holes right. It is not uncommon for CAD libraries to get it very wrong. (No kidding.) There's nothing worse than getting a beautiful new PCB back and finding that you can't insert the components! So, check and recheck.

When routing tracks around pads, ensure that there is sufficient clearance, as shown in Figure 4-4. Tracks should always change direction by 45-degree turns. Some PCB editing programs allow you to do a *design rule check* (also known as an *electrical rule*

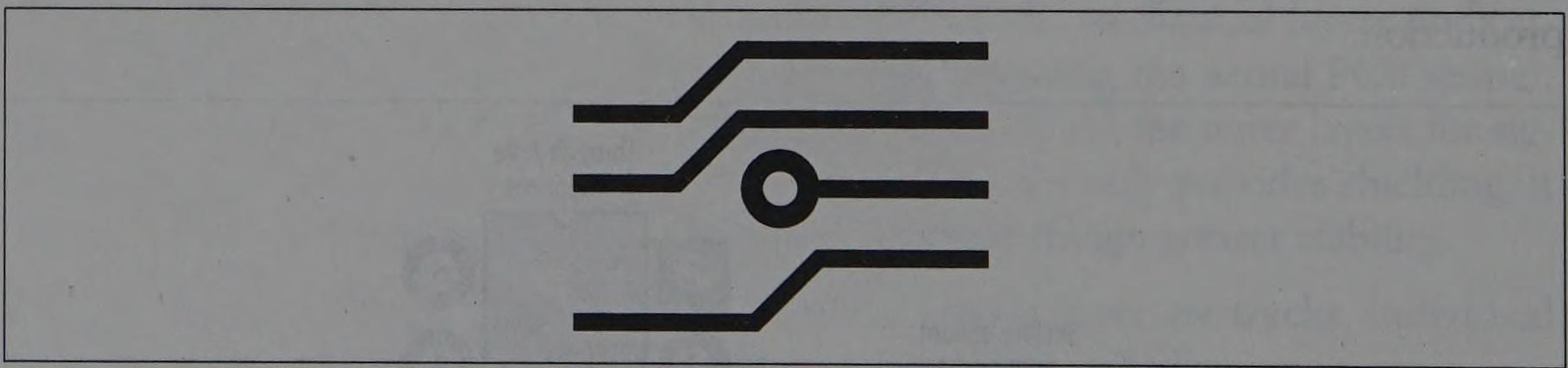


Figure 4-4. Routing tracks around a component pad

check) to ensure that correct clearances are maintained and that there are no potential shorts. (It's no guarantee that there won't be a problem, but it's a start!)

Avoid right-angle turns (A) and close passes (B), as shown in Figure 4-5.

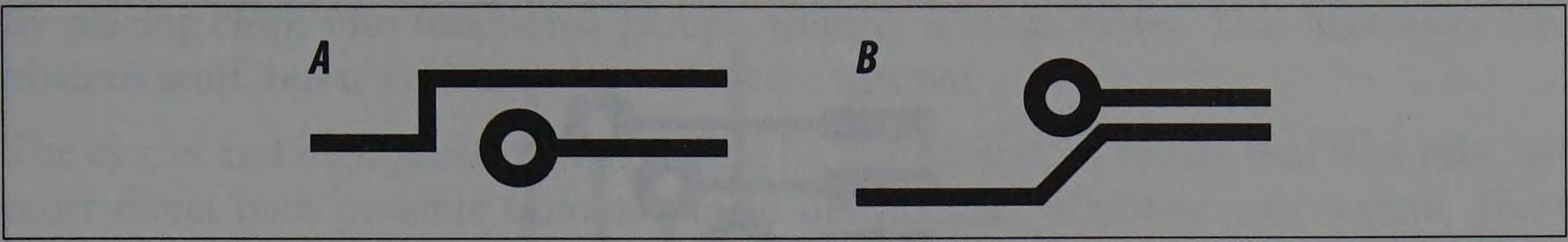


Figure 4-5. How not to route tracks around pads

Closely spaced pads on surface-mount components can present a problem. Often the tracks leaving a surface-mount device are too close together to actually do anything with. The solution is to *fan out* the tracks, thereby giving greater spacing to the tracks. This is shown, in a simplified form, in Figure 4-6.

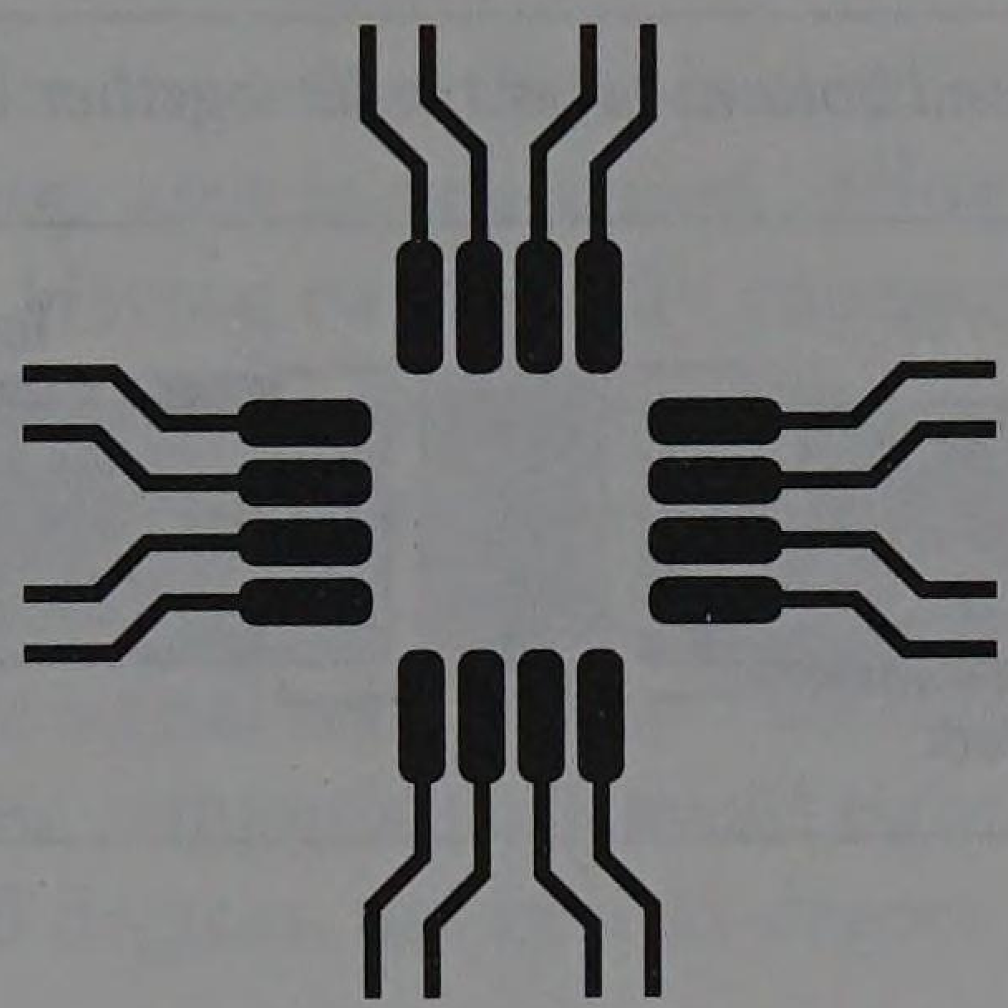


Figure 4-6. Fan out from a surface-mount device

Vias are used to connect tracks on different layers together (Figure 4-7). They are, in effect, little pads. Vias can either be through-hole vias appearing on all layers (Figure 4-8) or *blind* or *buried vias*, appearing only on the layers they are interconnecting and intermediate layers. Making the vias as small as possible aids in routing the PCB, but check with your PCB manufacturer as to how small you can go. Remember to ensure that the outside diameter of the via is sufficiently bigger than the hole, so that the entire via is not drilled out during fabrication. If space permits, a useful trick is to make the vias with 0.4mm holes. That way, if there is a bug in the PCB layout or a manufacturing fault, you can use the vias to solder in wire-wrap wire, and manually make (or remake) a connection.

Fills are used to provide shielding to certain sections of the PCB and also for circuit paths that carry a lot of current. Ground fills are commonly placed in and around analog sections of the circuit to isolate them from digital crosstalk.

Laying Out a PCB

The first thing to note when laying out a PCB is that someone (or some robot) is going to have to assemble it. As tempting as it might be to cram everything into the smallest space possible, remember the limitations of whomever (or whatever) will be building it. That's not to say you should make the PCB as big as possible; just be

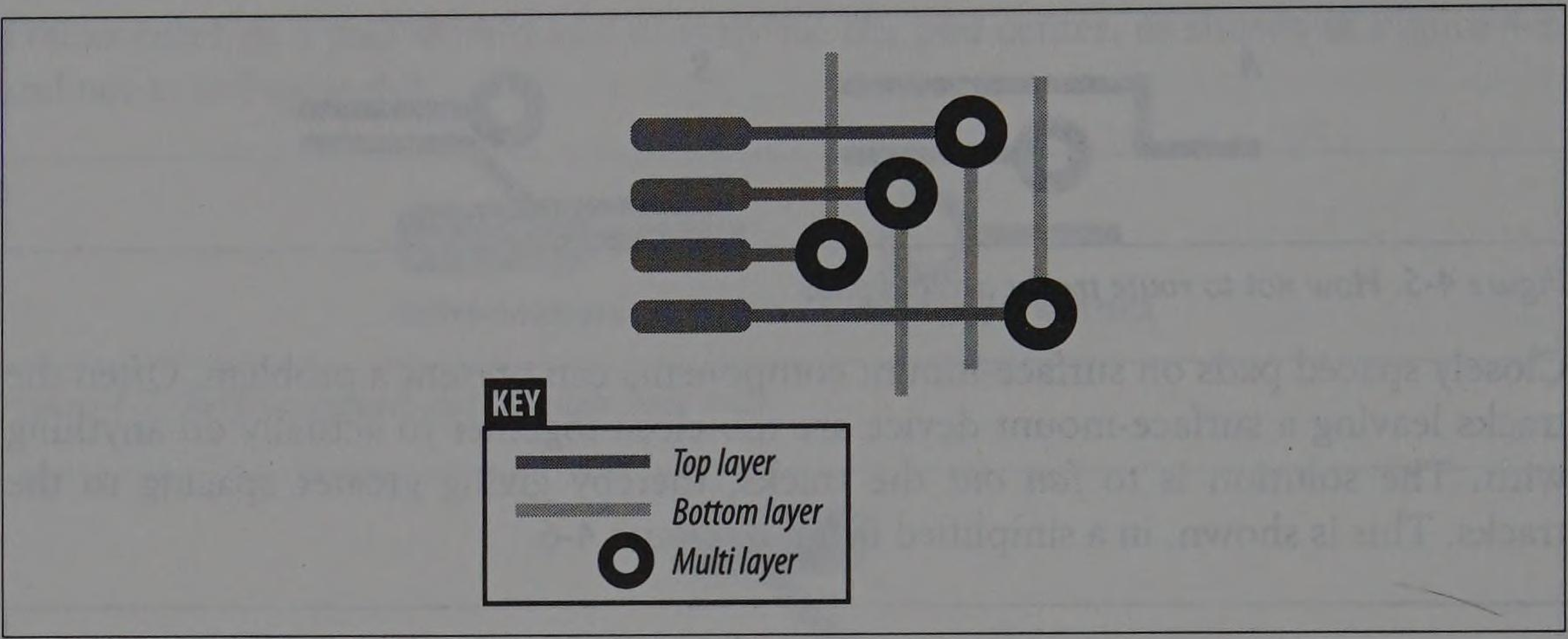


Figure 4-7. Vias connecting top- and bottom-layer tracks together

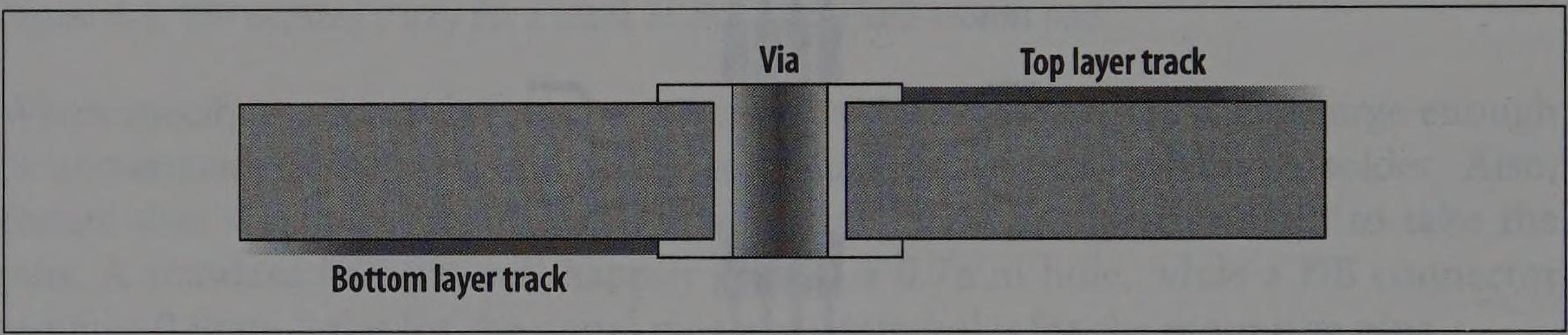


Figure 4-8. Via in cross section

realistic. Also, don't bring components and tracks right to the edge of the PCB. Leave a spacing of 5 mm (200 mils) around the outside. If your PCB is to go inside a case, or be mounted in some way, make sure that you have the dimensions correct, and don't forget to add mounting holes.

There are two schools of thought in placing components (especially integrated circuits) on a PCB. The first is that all components should be placed with the same orientation. For instance, the pin 1 of each chip should point to the upper-left corner of the PCB. This simplifies populating (or *stuffing*) the board with components, especially if this is to be done by a contract manufacturer in an automated process. Having varying orientations may add to the expense of production, if you're not hand assembling the boards yourself. Some people also think that this makes a board look neater.

The second school of thought is that you orient the chips so as to optimize the routing process. The pinouts of different chips are not necessarily conducive to uniform orientation, and spinning one chip 90 or 180 degrees to its neighbor may greatly simplify the routing of tracks between the two. This can lead to a smaller board size, fewer vias, and shorter track lengths. This then results in lower PCB cost, less noise, less crosstalk, and better noise immunity, which is especially important in higher-speed systems.

Whatever you decide about orientation, group related components together. Put the voltage regulator and its support components near the power connector. Any analog circuitry (such as sensors or amplifier circuits) should be as far from this as possible.

By placing chips into functional groups, routing is made easier. This may seem like obvious stuff, but it's amazing how often it's ignored.

The clocks and high-speed signals should be routed first, to ensure that they take the most direct path possible from source to destination. Wherever appropriate, place shielding (fills) to isolate these signals from other parts of the PCB. This should be done prior to routing other connections; otherwise, there may not be sufficient space later on. In particular, tracks should never be routed under or around crystals, oscillators, or any clock generation circuit, and these components should be isolated by fills (connected to ground) from the rest of the circuit. Crystals should lie flat against the PCB (rather than being mounted vertically), and a ground plane should be placed under them to shield from emissions.

For high-speed signals, make sure that there is a ground return path close to the track so that the current loop area is minimized. Allow as much space as possible between high-speed tracks. Having two rapidly changing signals in close proximity will result in crosstalk, and this will cause unreliable operation. Every track has an inherent impedance (resistance); although small, it can affect the transmission of fast signals. In particular, a via or sharp corner represents a change of impedance along the track, and this can cause signal reflections. Therefore, it's important to keep the number of vias to an absolute minimum and avoid right-angle turns in tracks. If you need to make a track turn 90 degrees, use two 45-degree turns in succession.

In high-speed systems, you need power and ground planes that are continuous. In other words, you need planes that cover the entire PCB with no breaks. Any break in the power or ground plane makes the current loop area larger, and this can increase inductance and radiation. This means, for high-speed systems, you really need to use four or more layers on the PCB. For low-speed microcontrollers, you can get by without separate planes or by providing fills in and around components on the signal layers.

When routing buses (such as data and address), keep the tracks running parallel if possible (Figure 4-9). This is bad practice for clock signals, since it can induce

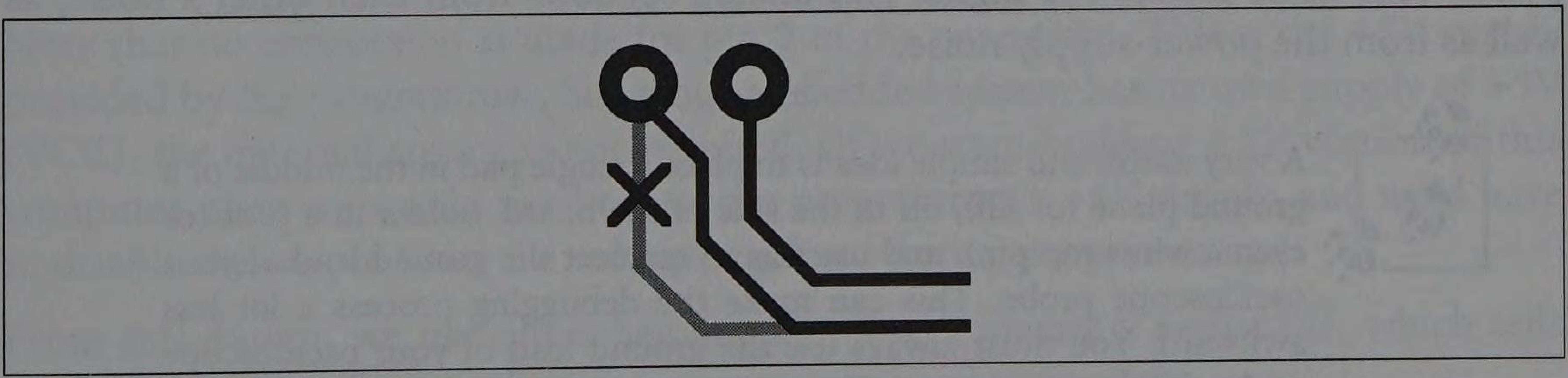


Figure 4-9. Keep buses parallel to minimize skew

crosstalk in neighboring tracks, but is appropriate for buses. The reason is that bus signals will change state together and will then hold that state until the next transaction. The device receiving the bus signals will sample their state only when they are stable (unchanging). Since crosstalk is generated on change-of-signal state, running

parallel buses is not a problem. By keeping the bus tracks parallel, the signals travel approximately the same distance for each track. A track that takes a different path completely will have a trace-length mismatch, and this can increase signal skew, where the time it takes for a signal to propagate is shifted. This can adversely affect signal quality in high-speed systems.

Stubs are short tracks that leave a main track to connect to a component (Figure 4-10). A stub represents an impedance mismatch for a signal and can result in reflections. A better way is to place the component so that the pad lies on the primary signal track (Figure 4-11).

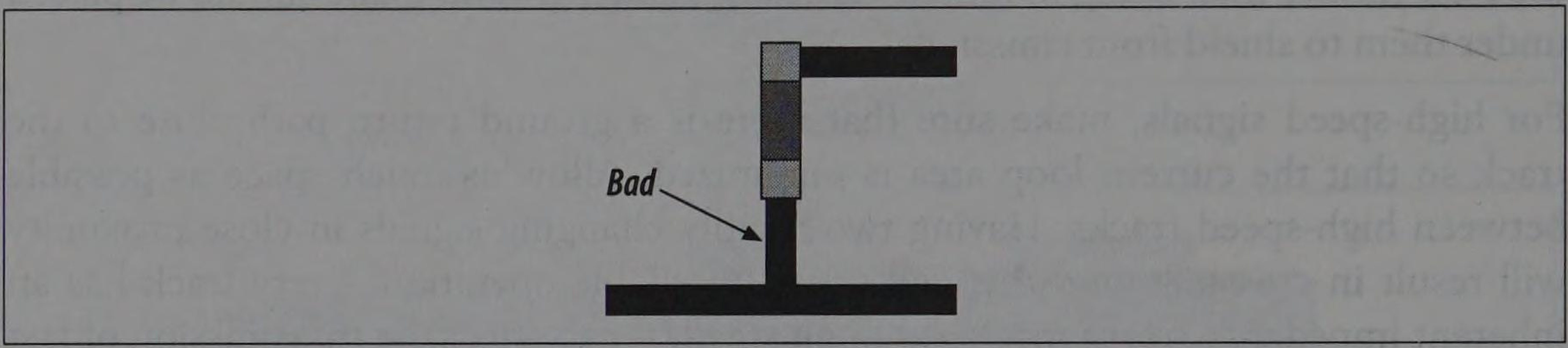


Figure 4-10. Avoid stub tracks

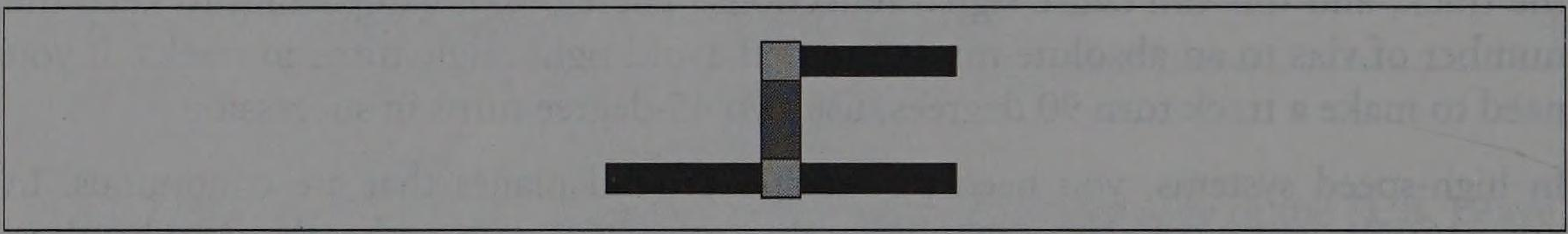


Figure 4-11. Place surface-mount components directly onto tracks, whenever possible

All power and ground traces should be as fat as possible, and if feasible, separate power and ground planes (layers) should be used. The power ground (ground coming in with the power supply) should be separate from signal ground or digital ground (the ground running to all your chips), and both separate from the analog ground, if one is present. They should all be connected together, but only at one point. This helps isolate the digital and analog sections from each other's noise, as well as from the power-supply noise.

> A very useful and simple idea is to place a single pad in the middle of a ground plane (or fill), off to the side of the board. Solder in a post (or even a wirewrap pin), and use this to connect the ground lead of your oscilloscope probe. This can make the debugging process a lot less awkward. You must always use the ground lead of your oscilloscope probe. Without it, you can't get an accurate picture of the timing and voltages of your signals. (Voltage is the potential difference between two points, so you must have a reference.) It's very important.

Decoupling capacitors should be as close as possible to each power pin of each integrated circuit. Figure 4-12 shows two components on a two-layer PCB, with power

and ground tracks routed horizontally through the middle and decoupling capacitors placed close to the power pins.

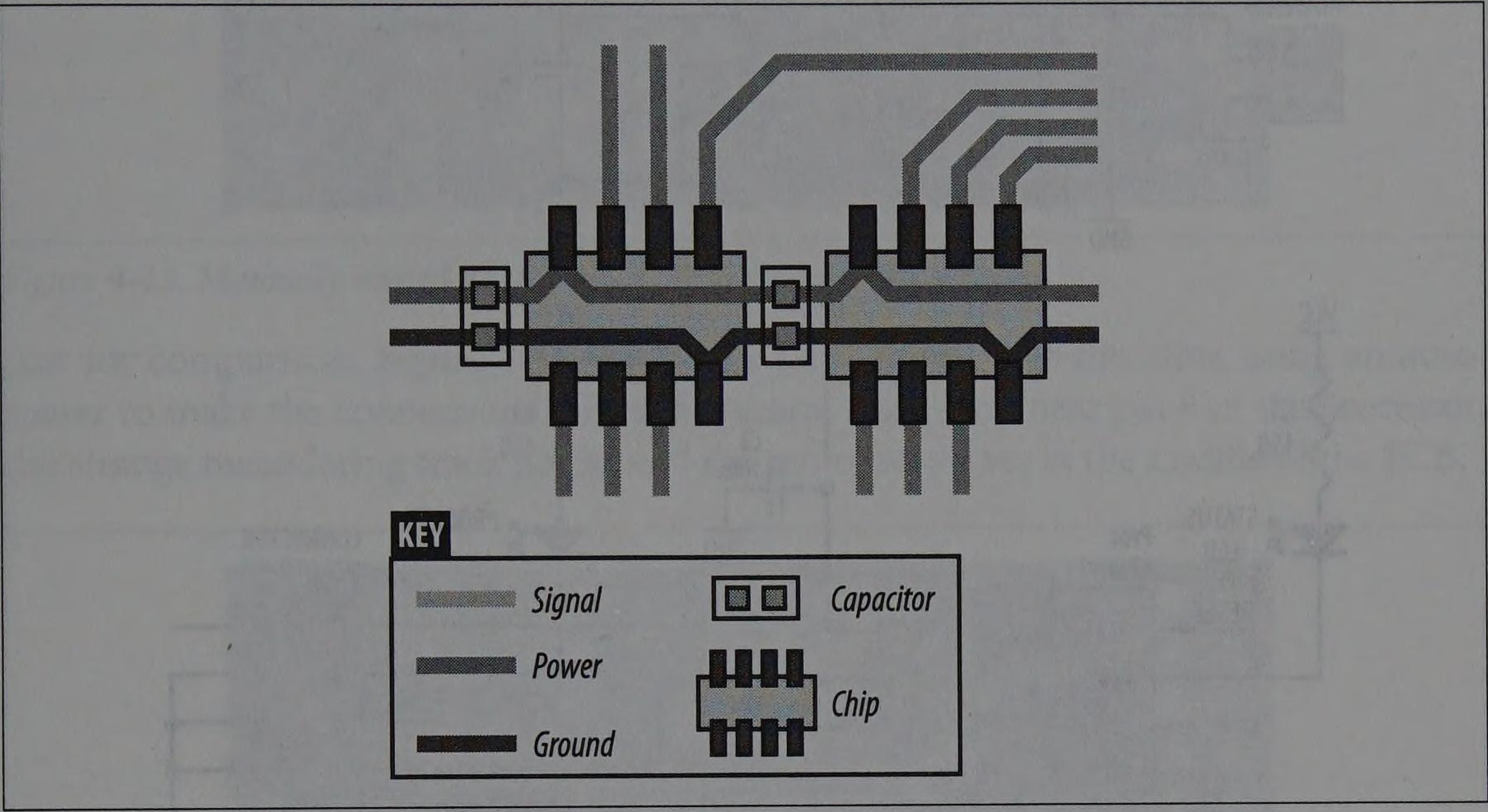


Figure 4-12. Decoupling capacitors should be placed as close to the chips' power and ground pins as possible

Routing a Design

So, with all that in mind, let's create a simple circuit board for an AVR computer. This design is covered in detail in Chapter 6, but for the moment, we'll just use it as an example so that we can see the process of producing a printed-circuit board.

We start with the schematic (Figure 4-13). This design brings together the voltage regulator circuit, the AVR processor with a status LED, and the in-circuit programming interface.

Note that no connection is made for pin 2 of the connector. This is the +5V supply provided by the programmer. Since our embedded system has its own supply of +5V (**VCC**), the external source is not required. (If we were building a 3V version of this computer, then we would need to use the programmer's +5V supply, and we'd have to disable the output from the voltage regulator during programming.)

From this design, we use our schematic editor to generate a netlist file, which tells the PCB editing software what interconnections need to be made.

Importing the netlist file into the PCB editor automatically loads the component footprints. These are manually rearranged to provide optimum placement, ensuring shortest track runs between components (Figure 4-14). Note how related components—such as the voltage regulator, C1, C2, and the power connector—are placed together. The silkscreen (overlay) layer shows the outline of the components. In this example, sur-

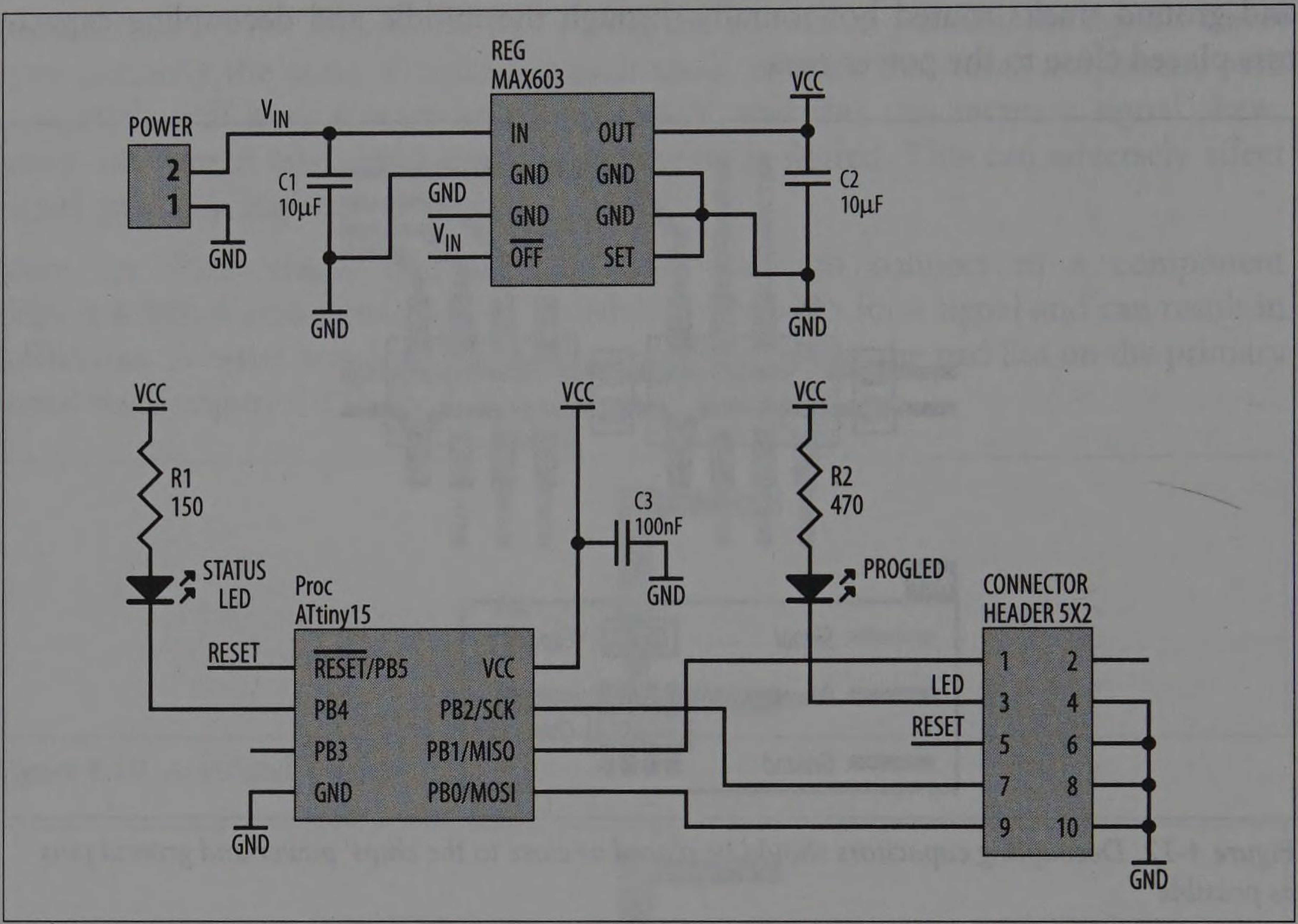


Figure 4-13. The full schematic, including connector for I/O and programming

face-mount components have been used for the resistors, capacitors, and LEDs, while the two integrated circuits are DIPs. Note the three-pad triangular LEDs. Only two of the pads are connected to the internal LED; the third is unused. This tiny circuit board measures just 2" by 0.6".

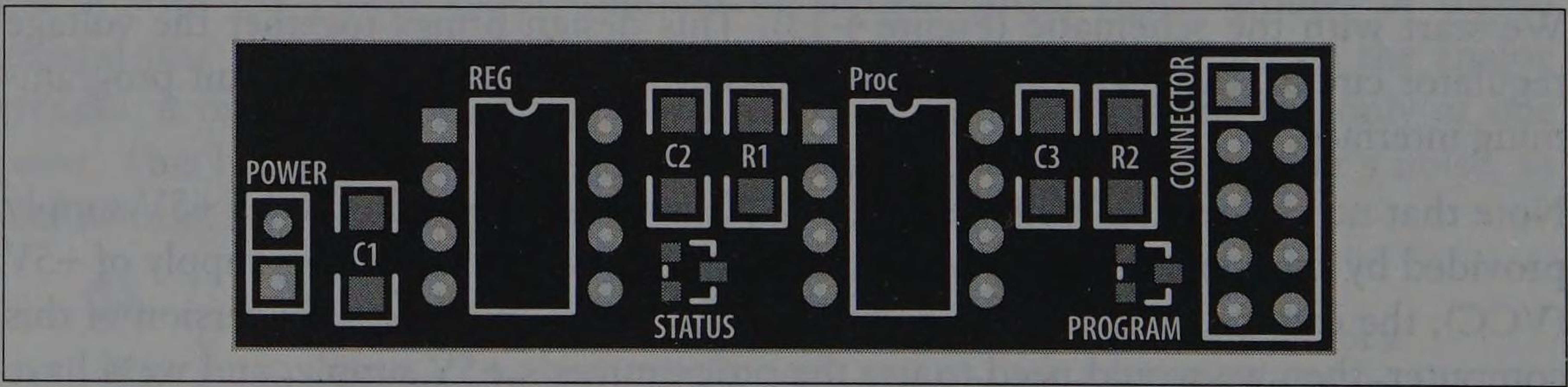


Figure 4-14. Components placed onto PCB

Once you're satisfied with the component placement (and this may need tweaking as you go), the connections are routed. Figure 4-15 shows the PCB with manually routed connections. In this case, the overlay layer has been "turned off" for clarity. Note the use of fills for power and ground connections. For such a simple circuit, operating at low speed with no external system buses, the PCB layout is relatively trivial.

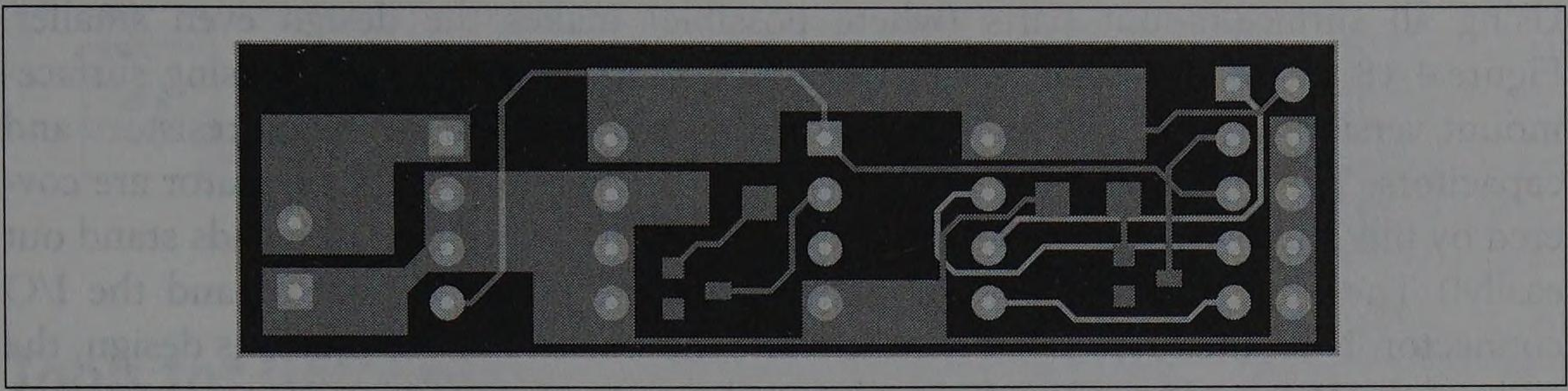


Figure 4-15. Manually routed board, using fills for power and ground

Just for comparison, Figure 4-16 shows the same board, but this time using an autorouter to make the connections. Note the bizarre track loop near pin 8 of the processor, the strange meandering track paths, and the unnecessary via in the middle of the PCB.

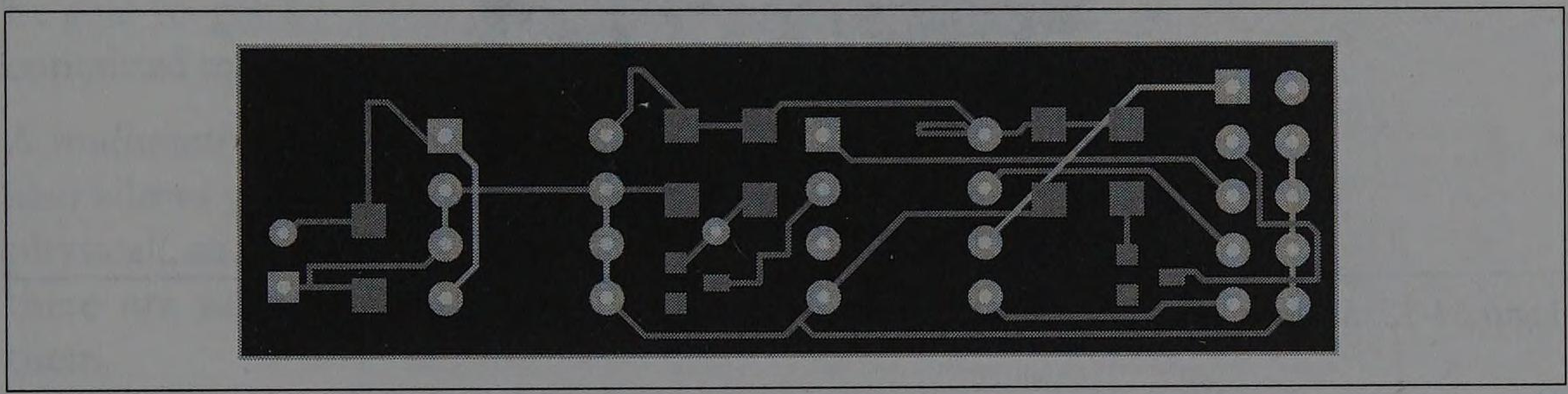


Figure 4-16. The psychedelic results of using an autorouter

This is a simple board, so even an autorouter can make most of the connections. On complex boards, the average autorouter gives up about halfway through, after first making a complete mess. There are autorouters that do a much better job than this, but they are very expensive.

For greater noise immunity, a polygon plane is placed on the bottom layer of the manually routed PCB to act as a Faraday shield (Figure 4-17). Note how the polygon has "flowed" around the component pins, yet has connected to the ground pins. In this way, the polygon is a ground plane, providing a (small) degree of noise immunity for the system. The "void" region in the middle right is where it could not reach, due to the prerouted tracks. If designing a four-layer board, the polygon fill would be placed on a separate layer and should have no discontinuities at all, except where it flows around the pads of through-hole components.

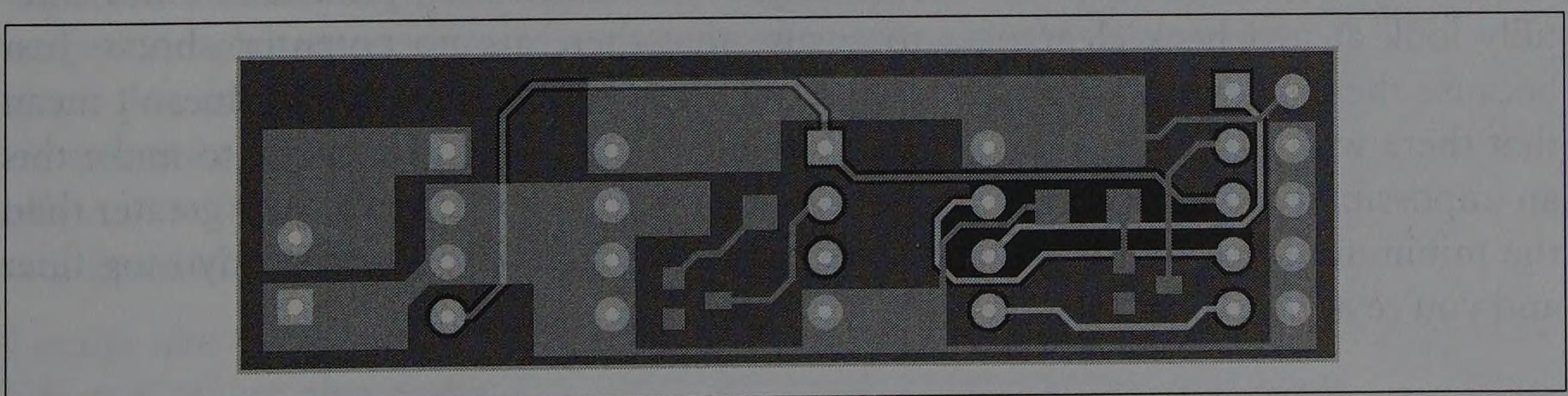


Figure 4-17. Manually routed PCB, with Faraday shield

Using all surface-mount parts (where possible) makes the design even smaller. Figure 4-18 shows the same design (manually routed), but this time using surface-mount versions of the processor and regulator and even smaller-sized resistors and capacitors. The new board size is just 1" by 0.5". The pads for the regulator are covered by fills and so are not apparent. (Once the PCB is fabricated, the pads stand out easily!) The only through-hole components are the power connector and the I/O connector. Note the four vias needed to route the tracks. For the previous design, the pads of the DIP components effectively acted as vias. On the all-surface-mount version, since just about everything is on the same layer, vias are required to take tracks to the bottom layer of the PCB.

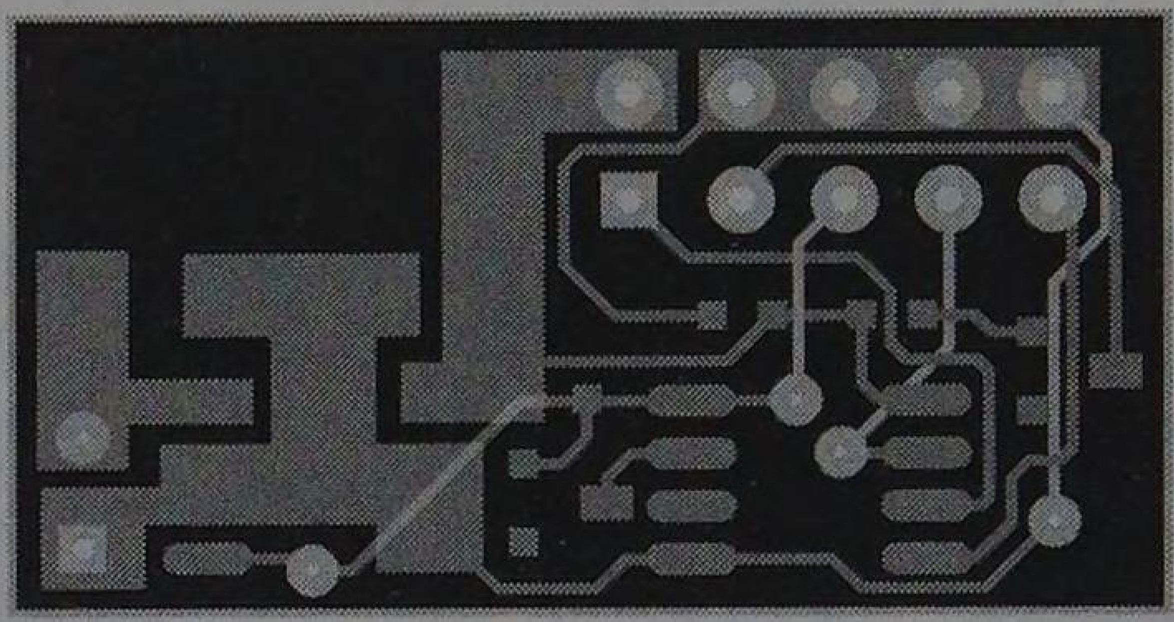


Figure 4-18. Surface-mount PCB

You could make this board even smaller (perhaps 0.5" by 0.5") by placing surface-mount components on both sides of the PCB. This adds to the cost of construction if you're having it professionally done.

Figure 4-19 shows the surface-mount PCB, now with a Faraday shield.

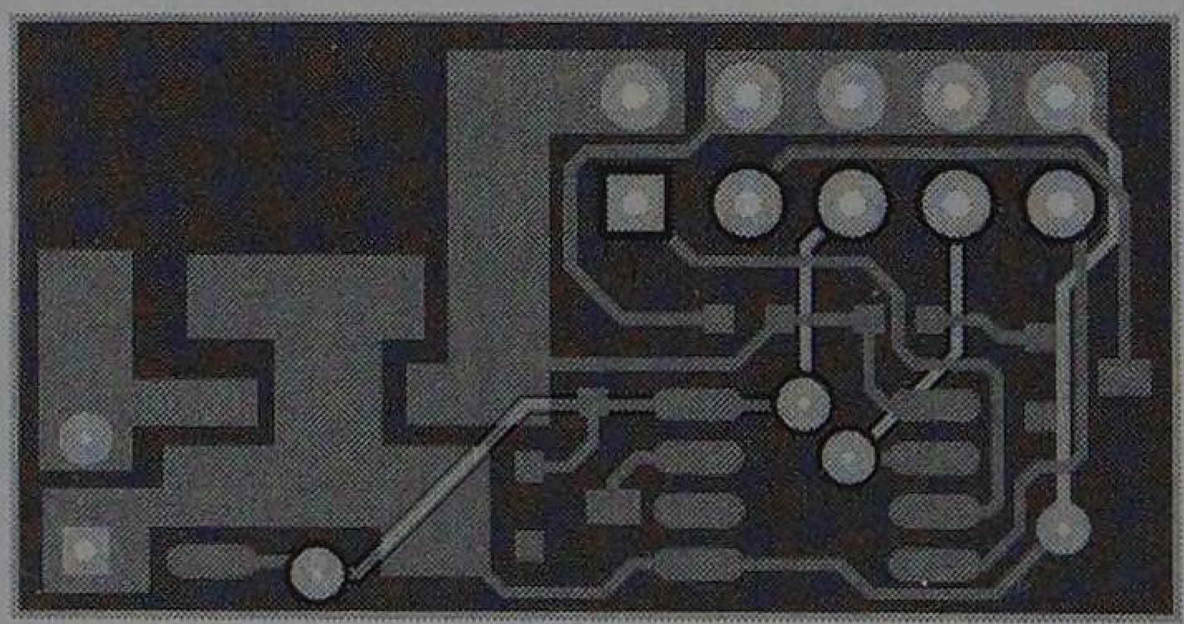


Figure 4-19. Surface-mount PCB, with Faraday shield

Before sending off your circuit board design to be fabricated, print it out and carefully look at it. Check clearances to ensure that there are no potential shorts. Just because there's a whisker gap between a track and a via on the screen doesn't mean that there won't be short there when it's made. Give enough clearance to make this an impossibility. Good practice is to set your clearances to be equal to or greater than the minimum track width to which the PCB manufacturer can etch. Anything finer and you're asking for trouble.

When your design is printed out, place the physical components on the paper and check for clearances. Just because the component outlines in the CAD package didn't collide does not mean that the physical chips won't. It is much easier to solve these problems before the PCB is fabricated than after.

Tools for Debugging

For the systems described in this book, the minimum debugging tools you will need are a multimeter and an oscilloscope. *Logic analyzers* are very expensive tools that allow you to monitor and diagnose digital signals. They are essential for developing high-speed and complex systems (especially those with buses), but you should be able to get by without them for the designs in this book. Certainly, for a self-contained microcontroller, a logic analyzer is of no use at all.

A *multimeter* allows you to measure current and voltage, but more importantly, it also allows you to do a continuity test between two points (and verify that there is a physical, and therefore electrical, connection). However, do *not* do continuity tests if there are sensitive components in your system. The continuity test may damage them.

Don't assume that just because a signal is present at one end of a trace it is present at all points along the trace. *Check everywhere* with an oscilloscope probe, and use your multimeter to confirm that signal paths are connected properly.

The *oscilloscope* allows you to view waveforms within your system, and as such, it is your principle debugging tool. Oscilloscopes range from the crude and ancient to the expensive and sophisticated. While you don't need to spend $100k on an oscilloscope, you will need an oscilloscope that can accurately view waveforms. That rules out the $20 antique you picked up from Mr. Gorsky's garage sale down the road.

You will need an oscilloscope of sufficient bandwidth to view the signals within your computer. There's no point using a 20MHz oscilloscope to look at a 100MHz system clock. The oscilloscope simply won't see it and, therefore, neither will you. The higher the bandwidth, the more you will see. While you may think that a 4MHz embedded processor might not need a 100MHz oscilloscope, that oscilloscope will allow you to see the rising edges of the waveforms as rising edges (and not just vertical transitions) and view minuscule timing differences that may be having an adverse effect. It will also allow you to see fine spikes of noise or ringing on your signal lines, which may be adversely affecting your machine.

I really like the low-cost Tektronix oscilloscopes for debugging embedded systems. HP and others also make nice tools. If you're serious about developing embedded

hardware, it's worth investing in one. Keep an eye out for startup companies going under—you may be able to pick up some great test gear going cheap!

> When using an oscilloscope, it is critical that you connect the ground clip of the probe to a ground connection close to (or better, *on*) your embedded system. Without this, your measurement of the signal will be affected by ground loop problems, and you will not get an accurate reading. You'll spend ages chasing phantoms, all the while missing the real problem.

Another development tool is the *In-Circuit Emulator* (*ICE*). This is a small module, with the same footprint as the processor, which is placed into the target system under development. Under the control of software executing on a PC and emulating the embedded processor, the ICE behaves just as the processor would in-circuit. This allows you to interactively debug your hardware and software. This can be especially useful in systems based upon self-contained microcontrollers, where it would be otherwise difficult (impossible) to get to the system internals.

Some ICEs are better than others, and as with everything, you get what you pay for. For really sophisticated tools that closely match the timing and electrical characteristics of the processor, expect to pay big bucks. Cheaper systems will emulate the processor's operation, but will do so with completely different signal timings. Also, for each processor type around which you develop systems, you'll need a different ICE.

Some engineers use ICEs heavily during their embedded system's development process. Call me a heretic, but I get by quite well without them. The catch with an ICE is that no matter how good a particular tool is, it is never going to be *exactly* like the real thing. There will always be some slight difference in the electrical characteristics or in the timing. The engineers at Boeing have a saying: "Test what you fly; fly what you test." In other words, there's no substitute for the real thing.

Putting It All Together

Once the PCB has been fabricated and checked carefully to ensure that all pads and tracks are intact and properly etched, do the construction a step at a time and check everything as you go. Do a continuity test between the ground pads and the ground pin on the power connector.

Start construction by soldering in the power connector and the voltage regulator and its support components, including the power LED if you've included one with your regulator.

Soldering

Soldering is very easy to do well and very easy to do badly. The basic skills are easy to learn. Becoming a wizard with the soldering iron is not hard to achieve.

Safety First

The most important thing to note about solder is that it contains lead. Therefore, all soldering should be done in a well-ventilated work area, and you should avoid breathing the fumes! After soldering, wash your hands, especially before eating! Solder can splatter, so always wear protective eyewear and clothes.

Solder is a metal alloy with a relatively low melting point. It is used to bond components to circuit boards and forms a conductive join. There are two basic categories of soldering tool—the standard soldering iron and the rework station. Rework stations blow heated air through a small nozzle and are primarily used with surface-mount components. However, it is relatively straightforward to solder all but the finest of surface-mount components using a standard soldering iron. You don't necessarily need the more expensive rework stations.

The key to soldering well is to control the heat and the amount of solder that flows onto component pins. Too much heat can damage a component (especially sensitive integrated circuits) and can overheat solder as well. Read the datasheets to determine the maximum temperature (and duration) that the components can take, and ensure that your soldering iron does not exceed that. Variable-temperature irons allow you to set the temperature, thereby avoiding overheating. The tip of your soldering iron should be thin, allowing you to do fine work. An old-style iron with a large, bulky tip (intended for electrical work) is not appropriate for soldering electronics.

Whenever you solder your PCB, make sure that it is not powered! The tip of a soldering iron is grounded, and touching this to a pad with volts on it is *not* a good idea!

Similarly, when inserting or removing socketed components, ensure that the system is powered down. Most semiconductors do not appreciate being plugged into a live system.

There should be enough solder to make a good contact, but not so much that it bulges up or, worse, shorts a neighboring pin (Figure 4-20).

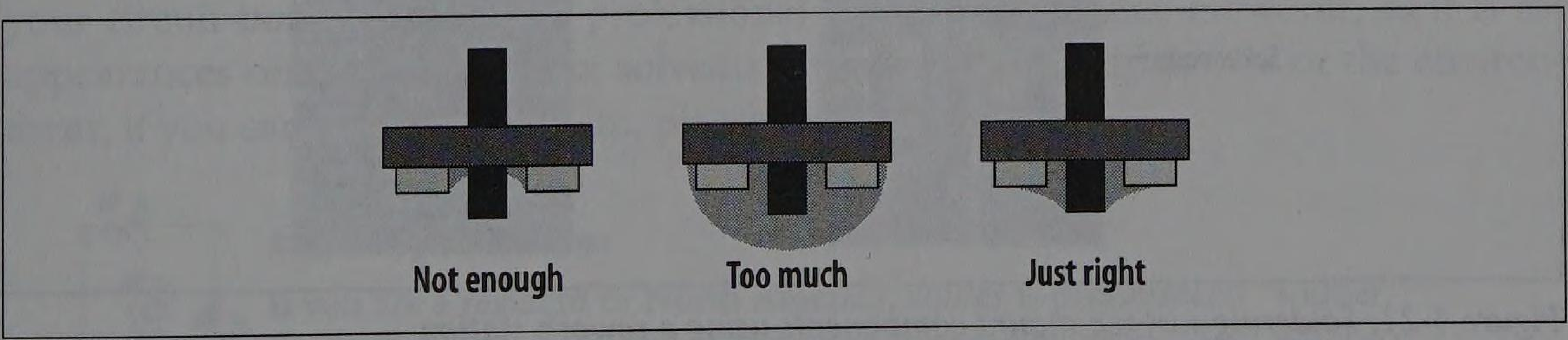


Figure 4-20. Component pins soldered to a PCB

During the Apollo/Saturn missions, NASA found that teaching their technicians the correct way to solder saved them several hundred pounds in takeoff weight.

When soldering through-hole components (such as DIP-packaged chips or connectors), place the component into its hole and ensure that it is mounted correctly and sitting flat. To begin, solder one pin only, then check that the component is still seated correctly before doing the remaining pins. With the iron in one hand and a thin strand of solder in the other, bring the two together so that they meet at the pin to be soldered. Within a second, the solder will flow around the pin and you will have a good join. As soon as the solder begins to flow, remove both the iron tip and the solder strand.

Common mistakes when soldering are to heat the component pin for several seconds before applying solder (causing the component to become too hot) or to apply the solder directly to the iron and then dab the molten solder onto the pin.

Soldering surface-mount components requires a different procedure. If you're using a rework station, you will need to use solder paste. This is sold in a large syringe. Solder paste easily dries out inside the syringe, so ensure that you seal the end when it is not in use. Before soldering a surface-mount chip, place a thin squirt of solder paste along each row of pads on the PCB. Too much paste can flow under the chip and short out when you solder, so keep the application light. You can always add a small quantity later. Place the chip onto its PCB pads, ensure that it is lined up correctly, then use the rework station to apply heated air (Figure 4-21). Too much airflow will either shift the chip off its correct orientation or, worse, blow solder paste underneath. Since solder paste is electrically conductive, this is not a good thing. Too little heat will result in poorly soldered joints, whereas too much heat can easily overheat and damage the chip. It is something of an art to get it just right, and so it's best to do considerable practice before tackling the real thing.

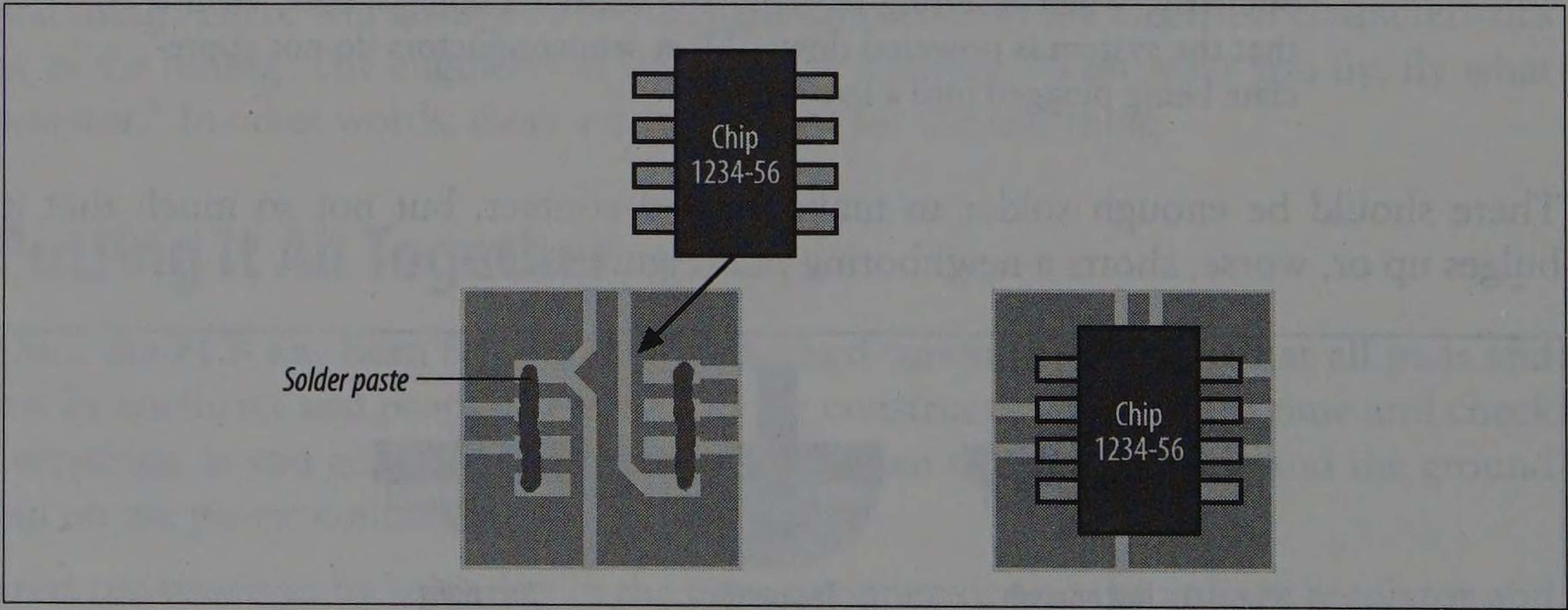


Figure 4-21. Soldering surface-mount components using a rework station

Surface-mount chips can also be soldered using a standard iron, although it's not recommended for really finely spaced chip pins. Unlike the technique with the rework station, solder paste is applied after the chip is in place. To begin, before putting the chip on the PCB, use the iron and either strand solder or solder paste to place a small

dab of solder directly onto one of the pads where the chip is to be mounted. Place the chip in position, aligning it carefully, and then use the iron to heat the pin resting on the solder dab. The dab will melt and fix the chip in place. Check the alignment again to ensure that the chip did not shift. If it did, reheat the pin again, and carefully shift the chip as appropriate. Once you are happy with the alignment, place a thin squirt of solder paste down each row of pins and as far from the edge of the chip as possible. Too much paste will flow between the pins, creating shorts, so keep it light. Gently and quickly run the tip of the soldering iron down each row of pins. The solder paste will melt and flow as you go and bond the chip to the PCB (Figure 4-22).

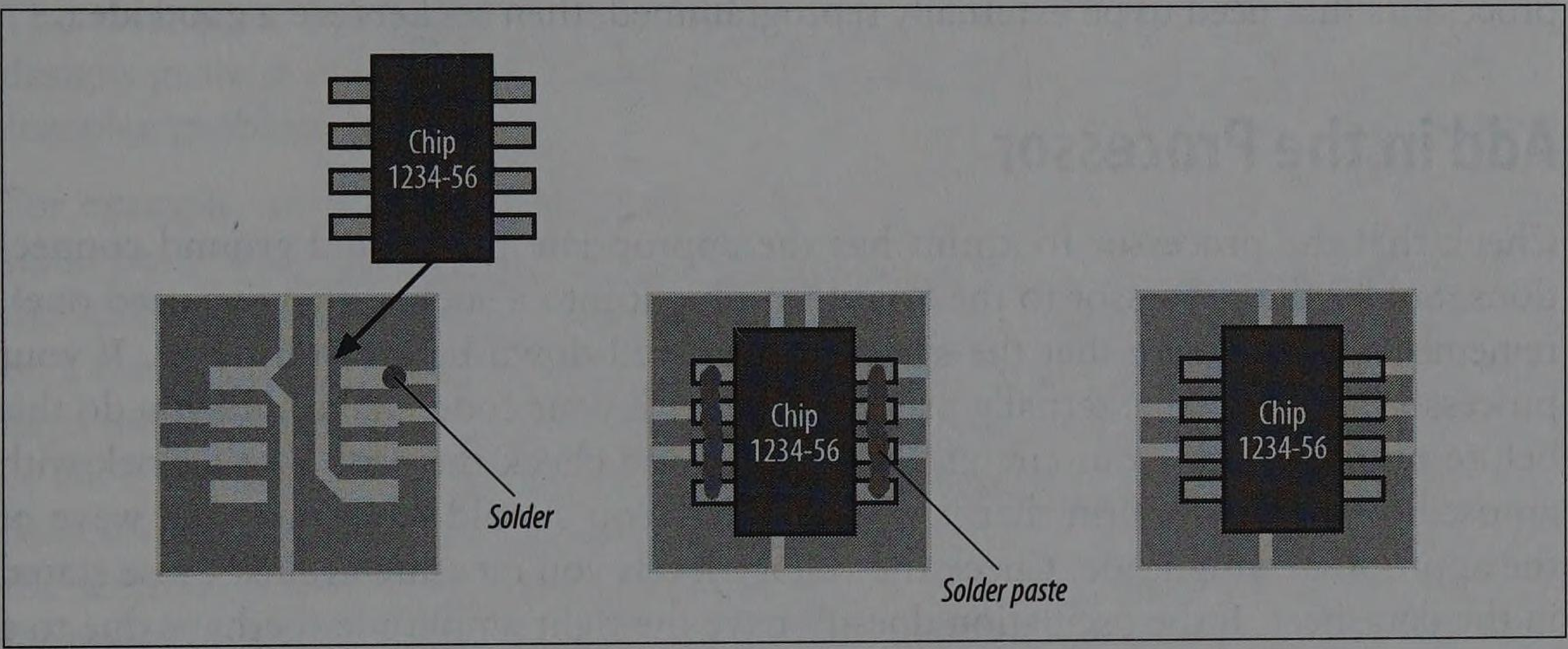


Figure 4-22. Soldering surface-mount components using a standard soldering iron

Solder is a metal alloy and incorporates a flux to assist flow. When heating the solder, it is common for the flux to separate and flow out onto the surrounding PCB, leaving a thin brown residue. Excess flux can be removed using special solvents, available from most electronics hobby stores and suppliers. Flux removers can be nasty stuff, so keep them away from skin and plastics and use in a well-ventilated work area. Flux residue is removed for cosmetic reasons only, and this will make your circuit boards look more professional to your customers. However, as it is for appearances only and since flux solvents are not good for either you or the environment, if you can avoid using them, please do so.

A note on pronunciation

If you are a resident of North America, solder is pronounced "sodder." If you live anywhere else in the English-speaking world, you will pronounce it as "sol-der." So, Americans, be advised that if you say the word as "sodder" to non-Americans, they may not know what you're talking about. Instead, they make think you're confessing to strange and unspeakable acts, rather than talking about bonding metals together.

Powering Up

Once you've soldered in the components needed for the power supply, power up the board and check that this is operational. Also check that you have power on every pad on the board where you expect power to be, and check the ground pads to make sure that there is no power where you expect no power to be.

Next, solder in the power-decoupling capacitors for the ICs. Add in the processor's oscillator and decoupling capacitors. If the oscillator is a module, check its operation with an oscilloscope. Does it have a waveform on its output pin?

If IC sockets are used, solder these next, then insert the components. If you're using processors that need to be externally reprogrammed, then sockets are a good idea.

Add in the Processor

Check that the processor footprint has the appropriate power and ground connections. Solder the processor to the board (or plug it into a socket if you've used one), remembering to ensure that the system is powered down before you do so. If your processor needs to be externally programmed with your code, make sure you do this before putting it into your circuit. Power it up and check the processor's clock with an oscilloscope to confirm that it is oscillating. You should see a nice sine wave of the appropriate amplitude. Check the voltage levels you measure against those stated in the datasheet. If the oscillation doesn't have the right amplitude (perhaps due to a poor connection or a partial short), it may not be able to drive the processor.

If the system you are building uses a microcontroller with no external ROM (such as the example presented in this chapter), the first test software you will write will simply waggle an I/O line of the processor. Observing this with an oscilloscope will allow you to see if your system is executing code correctly. If you included a status LED in your design, turn it on! Seeing a status LED blink on for the first time on a machine you've designed and built yourself is sure to bring a smile to your face.

Once you've confirmed that the processor is operating under software control, you can begin to add in the other hardware and software components of your application. A word of advice, though: don't get too adventurous at any stage of the building process. If everything suddenly stops working, it's much easier to find the cause if you've made only one change or addition. Take things a step at a time.

Some Thoughts on Debugging

Debugging is as much an art as a science. You can load a workbench to breaking point with all sorts of expensive test equipment, yet without a logical approach and a clear mind, elusive bugs will never be found. Conversely, by "right thinking," the strangest of bugs can be isolated with a minimum of tools. While it is true that the

more complex the system under test, the harder it is to nail down a fault through detection, it is also true that the most advanced and useful debugging tool you have at your disposal is your own brain. Therefore, learning to debug is learning to think carefully and clearly.

Debugging hardware can be a lot trickier than debugging software. With code, you can always put in some diagnostics to inspect the execution. That's not to say that debugging software is trivial—far from it. But with hardware, it is often either a case of it all works, or nothing works. Software has the advantage of being able to be brought into operation gracefully. For hardware, you need to have an awful lot working right from the start.

The essence of debugging is establishing what works and what doesn't work. As designs grow in complexity, finding hardware and design faults can become quite a complex problem.

For example, your embedded system may not be outputting characters through its serial port. Why? Perhaps it's a bug in the code. Maybe there's a cable fault. Maybe the RS-232C interface chip is dead. Maybe the serial chip itself is dead. There may be a timing problem with the serial chip's oscillator or a voltage-level problem. Perhaps the processor itself is not coming out of reset and therefore not executing code at all. If so, maybe it's the power-on reset circuit failing to kick in or the brownout detector kicking in when it shouldn't. Maybe a data line between the processor and the serial chip is not connected, perhaps due to a manufacturing fault with the PCB. Or maybe it wasn't soldered correctly. Perhaps your voltage regulator isn't operating properly, or maybe you've a faulty power supply. And those are just the obvious causes that spring to mind. There are a thousand others lurking, with big teeth and a nasty disposition.

Any one problem may have a multitude of possible causes. Debugging is therefore about isolating a fault, and this is best done by a 20 questions approach. Use divide-and-conquer to solve the problem.

Let's take the example of the faulty serial port problem. You discover the problem when you first try to test the serial port. Your simple test code fails to output a character. Is the problem in software or hardware? If hardware, is the problem with the cable, the serial chip(s), or a more fundamental problem with the core system? Check the cable and the terminal (or host PC) first. Disconnect the cable from the embedded computer, and with a piece of metal (a screwdriver blade will do), short out pins two and three (**Rx** and **Tx**) on the cable connector. Now type something on the terminal (or the terminal software on the PC). What comes out of the terminal should echo back through the short and appear on the screen. That will tell you whether there is a cable fault and whether the terminal is set up correctly.

If that works, then the problems lie in your embedded system. Replace your serial test code with code that does something else that is simpler (like waggle a digital I/O line or flash a LED). That simple action will tell you volumes. (Archimedes once said,

"Give me a lever long enough and I will move the world." Well, give me a status LED and enough time, and I'll debug the world too!) It will tell you whether your processor is executing code correctly, which in turn shows that the processor and ROM (if a separate chip) have power and are communicating correctly. It shows that the reset circuit, brownout detector, oscillator, voltage regulator, address decoder, and other support logic are OK. If any of these are failing, then the processor will not be executing code and therefore that I/O line will not waggle or that LED will not flash. By that simple test, you have ruled out a plethora of possible faults.

If that test failed, you know to look elsewhere for the problem, such as checking the oscillator, reset, or voltage regulator for correct operation. Divide and conquer. If the test passed, then the fault lies with the serial chip. Most serial chips include some digital I/O that can be manually set (such as **RTS**). Write some test code that does this. This simple test will show whether you can talk to the chip. If the test passes, you know to look at either your character-output software or the RS-232 driver. If the test fails, then the problem lies in talking to the chip. Use an oscilloscope to check the chip select and other control signals going to the serial chip. Are they active? Are they reasonable? Write some software that continually "jams" a byte at a register in the serial chip. While meaningless to the serial chip, a continuous write of the same number allows you to observe the bus activity. So, your (pseudo) code to do this is:

```
        load    r1,#0x55         ; load %01010101
loop    store   serial_control   ; write it
        jump    loop             ; continuously
```

You will expect to see the preceding bit pattern on the data bus (and importantly on the appropriate pins of the serial chip) at the same time the chip select and write enable are asserted.

This will enable you to locate a problem with the processor writing to the serial chip. Alternatively, if you can demonstrate that you can write to the chip correctly, then the problem lies either in the software or between the serial chip and the serial connector. By using the divide-and-conquer approach, you can isolate where a problem lies. Devise tests to prove each aspect of system operation.

Often you will be faced with a bug that makes no sense. Something should be working, and it is not. Everything you check seems right, but the total system just isn't working. It can be very perplexing. You have made a common error—you have made an assumption. Somewhere, even though you may not be consciously aware of it, you have assumed that some little detail is correct, when in fact it is not. This is the hardest obstacle to overcome. When you say to yourself, "It should be working, but it isn't! It doesn't make sense!" then say to yourself, "There is still something I haven't checked." Go looking for it. If you can't find it, then you haven't looked hard enough.

When designing your system and laying out the PCB, remember that you will have to debug it. So, design it with debugging in mind. Include one or more status LEDs. These are invaluable for debugging embedded hardware. Sure, you can do a lot with a remote

Read the Electrical Specifications!

I once designed a system that operated on a 5V supply. It worked wonderfully. I then went to produce the same system operating on a 3.3V supply, by using 3.3V versions of the same parts. It should have worked. It didn't. All the timing was correct; all the voltages were correct. Everything I checked was right, and yet nothing. No activity, not even a trace of signal from anything. I couldn't understand it. There was nothing left to test.

I knew that the design and code were correct for they worked beautifully in the 5V version. It had to be something specific to the hardware in the 3.3V system. But what? The processor had power, the regulator was working, the oscillator was going, and the reset circuit was operational. It should have been executing code. Yet, even the simplest of test software failed to go.

Somewhere, there was an incorrect assumption I was making. It took me more than a week to find it, and it was as subtle as they come. In going from the 5V system to the 3.3V system, I had chosen the 3.3V version of the processor. What I had assumed (logically, but incorrectly) was that the brownout detector (built into the processor) was designed to work at the correct levels. You'd expect that, but it was wrong. The manufacturer of the processor, when producing the 3.3V chip, had changed the operating voltage of the device but had left the brownout detector unchanged. So, in the 3.3V processor, the brownout detector kicked in if the supply was less than 4.5V! Hence, the processor never came out of reset and therefore never executed code.

I had assumed, incorrectly, that everything about the 3.3V processor was designed to work at 3.3V. For correct operation, the 3.3V processor needed to have the (optional) brownout detector disabled. This was not explicitly stated in the datasheet, merely implied through careful reading of the electrical specifications.

The moral of the story: don't assume *anything*, and check *everything*. If it still doesn't work, you haven't done the checking carefully enough.

debugger (such as gnu's gdb), but you have to get the hardware working to a certain level before the debugger can be made to run. Status LEDs will help you get there.

You are also going to need to look at signals with an oscilloscope, so include a ground pin on your circuit board onto which you can clip. Also, make sure that you will be able to get an oscilloscope probe to every circuit trace on the board to examine what's going on. If you can't get to a track, you can't ensure that there's no problem with that particular signal.

So even at the design stage, think carefully about how you can test the subsystems and isolate problems and put the necessary support into your design.

In the next part of the book, we'll look at some embedded processors and how you design systems based upon them. We start, in the next chapter, with the Microchip PIC processor family.

Read the Electrical Specifications!

I once designed a system that needed a 5V supply. It worked wonderfully. I then went to produce the same system operating on a 3.3V supply, by using 3.3V versions of the same parts. It should have worked. It didn't. On the bench, it was dead: the voltages were correct. Everything I checked was right, and yet nothing. No activity, not even a trace of signal from anything. I couldn't understand it. There was nothing left to test.

I knew that the design and code were correct, for they worked flawlessly in the 5V version. It had to be something specific to the problem with the 3.3V system. But what? The processor had power, the reset line was working, the oscillator was going, and the reset circuit was operational. It should have been executing code. Yet, even the simplest test software failed to run.

Somewhere, there was something I had overlooked. It took me more than a week to find it, and it was as subtle as they come. In going from the 5V system to the 3.3V system, I had changed to the 3.3V version of the processor. What I had assumed (logically, but incorrectly) was that the brownout detector (built into the processor) was designed to work at the proper levels. That it wasn't, for it was wrong. The manufacturer of the processor, when producing the 3.3V part, had changed the operating voltage of the device but had left the brownout detector unchanged. So, in the 3.3V processor, the brownout detector kicked in if the supply was less than 4.5V! Hence, the processor never came out of reset and therefore never executed code.

I had assumed, incorrectly, that everything about the 3.3V processor was designed to work at 3.3V. For correct operation, the 3.3V processor needed to have the (optional) brownout detector disabled. This was not explicitly stated in the datasheet, merely implied amongst other listings of the electrical specifications.

The moral of the story: don't assume anything, and check everything. If it still doesn't work, you haven't done the checking carefully enough.

debugger (various that signal), but you have to get the hardware working to a certain level before the debugger can be made to run. Status LEDs will help you get there.

You are also going to need to look at signals with an oscilloscope. Include ground pins on your circuit board onto which you can clip. Also, make sure that you will be able to get an oscilloscope probe onto every signal. Fine-pitch parts can be hard to access. If you can't get to a signal, you can't ensure that it is working.

So even at the design stage, think carefully about how you can test the hardware and isolate problems, and put the necessary support into your design.

In the next part of the book, we'll look at some embedded processors and how you design systems based upon them. We start, in the next chapter, with the PIC processor family.

PART II

Embedded Processors and Systems

Part II takes a look at several microprocessors used in embedded systems, ranging from the very tiny to machines with significant processing power.

Chapter 5 and Chapter 6 introduce you to two microcontroller architectures, the Microchip PIC and the ATMEL AVR. Their internal architectures vary considerably, but from a hardware viewpoint, they are similar. These two processor families are so simple that building a computer based upon them is trivial, as you will see. In the AVR chapter, you'll also learn about bus interfacing, developing valuable skills that will carry over for the other processors presented in this book.

In Chapter 7, I'll look at processor architecture using the Motorola 68000 series as an example. The 68000 is a powerful, widely used midrange processor suitable for a variety of embedded and control tasks.

From there, I go on to look at an unusual, yet powerful, processor family, the Motorola DSP56800, in Chapter 8. These processors are ideally suited to computationally intensive applications since they are adept at executing complex algorithms quickly and efficiently.

Since this is a book about hardware, we won't look at instruction sets. The processor datasheets give good coverage of the instructions, or you may choose to write your software in C or Forth, rather than assembly. In either case, a detailed look at software is beyond the scope of this book. You might like to refer to Michael Barr's excellent book *Programming Embedded Systems in C and C++* and Mike Loukides and Andy Oram's authoritative *Programming with GNU Software*, both available from O'Reilly & Associates. These two books give embedded software far better coverage than I could do justice to here.

CHAPTER 5

The PIC Microcontrollers

Where a calculator on the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and perhaps weigh 1 1/2 tons.

—Popular Mechanics, March 1949

To start our exploration of microprocessor hardware, let's look at the basics of creating computer hardware by designing a small computer based on a simple 8-pin PIC processor, the Microchip PIC12C508. The same design principles apply to the AVR and many other microcontrollers. This PIC processor is so simple that building a computer based upon it is trivial, as you will see. I'll also take a look at a midrange PIC processor and show just what you need to do to design an embedded computer based on one. Before getting into designing computers, let's take a quick tour of the PIC architecture.

A Tale of Two Processors

In the late 1970s, General Instruments had a 16-bit processor, known as the CP1600. It has long since passed into extinction and is all but forgotten, losing out to the Intel 8086 and the Motorola 68000. The trouble with the CP1600 was that it had limited I/O capability, and so General Instruments designed a tiny companion processor to act as an I/O controller. The idea was that this controller could provide not only the I/O for the CP1600, but also, being a processor in its own right, it could provide some degree of intelligent control. This processor was called the *Peripheral Interface Controller*, or *PIC*. The CP1600 died a quiet death, passing gently into oblivion, but its little companion lives on. In the mid-'80s, the microelectronics division of General Instruments was spun off into Microchip, and the PIC processor was its core product. The PICs are widely used. They live in the controllers of Sony PlayStations, children's toys, consumer appliances, and industrial systems.

The original PIC architecture has only one accumulator (known as the *working register*, or *w register*) and 25 to 368 bytes of RAM in the original processors. The program counter's least-significant byte, the status register, and various control registers are mapped into the lowest part of the RAM space and may be accessed by standard memory move operations. The upper part of the RAM space is for data. Microchip refers to the RAM space as "registers" although they have limited functionality as true registers. They are primarily for data storage.

The processor has a stack that is fixed to a depth of between two and eight entries (depending on the particular processor) and is used solely for holding return addresses for subroutine calls and interrupts. There is a single register, known as the *FSR (File Select Register)*, which can act as an index register into the RAM space. Limited indexed addressing is available using the FSR, and it can also be used to implement a pseudostack for user data.

Apart from a few exceptions, the PIC has no external buses and is a self-contained computer within a single chip. Only limited expansion is possible using the processor's peripheral interfaces (SPI and I²C, covered in Chapter 9) or digital I/O ports. The PIC excels in applications in which size and power consumption are critical. Being able to drop a tiny computer system into a design is a great bonus, and it is ideal for battery-powered applications, since it can (almost) run off the field of a stray electron.

The PIC is also very robust. It takes a *lot* to kill a PIC. I had one customer who inadvertently switched power and ground on his PIC-based computer and left it that way for a week. At the end of it, the little processor was still operational (once powered the right way). Another time, we tested a PIC-based datalogger by attaching it to the Indian Pacific Express. This is a long-haul passenger train that goes between Sydney and Perth, crossing the deserts of central Australia. Unfortunately, during the trial, the Indian Pacific was involved in a serious rail accident. A signaling fault caused a commuter train to impact the rear of the express, completely demolishing the end carriages. The datalogger had been attached (externally) to the rear of the train. It had absorbed the full impact of the collision, and when recovered from the wreckage, the datalogger was still operating normally. PICs are tough little processors!

The PIC is very RISC-like in many respects. The architecture is Harvard, with separate data and code spaces. The data space is 8-bits wide, while the code space is between 12- and 16-bits wide, depending on the particular PIC family. The data space is mapped into multiple banks, including most control registers. With only one accumulator, banked memory, and limited addressing modes, a reasonable percentage of a given program can be spent simply shuffling data around, much more so than many other processors. The PIC excels in small-scale, simple applications. However, the lure of its ultralow power consumption sometimes means that it is pressed into service running some quite involved algorithms. Writing complicated software for the PIC sometimes feels as impossible as trying to solve a Tower of Hanoi puzzle that has only a single peg. It can be a challenge! Many a PIC programmer has wished for just a bit more memory and just a few more accumulators. The announcement by Micro-

chip of the new dsPIC architecture, which is a significant advance over the standard PIC, has been received with chortles of joy by PIC developers around the world.

The Microchip software development environment (MPLAB) provides an assembler, a simulator, and software for burning code into the processors. MPLAB is freely downloadable from the Microchip web site. A number of commercial C compilers are also available for the PIC, but there is no port of the gnu C compiler for it. (At the time of writing, there are rumors that the gnu compiler will be ported to the new dsPIC architecture.)

For many simple digital applications, a small microprocessor is a better choice than discrete logic, for it is able to execute software. It is therefore able to perform certain tasks with much less hardware complexity. So, let's see just how easy it is to produce a small, embedded computer.

Starting Simple

The PIC12C508 processor is a tiny 8-pin computer, designed for the simplest control functions. It can be used in any small application when you need to monitor digital inputs or turn something on or off. Its I/O pins could be used to synthesize a SPI or I^{2}C interface (Chapter 9) or to control a motor (Chapter 12).

The processor's internal program address space is shown in Figure 5-1.

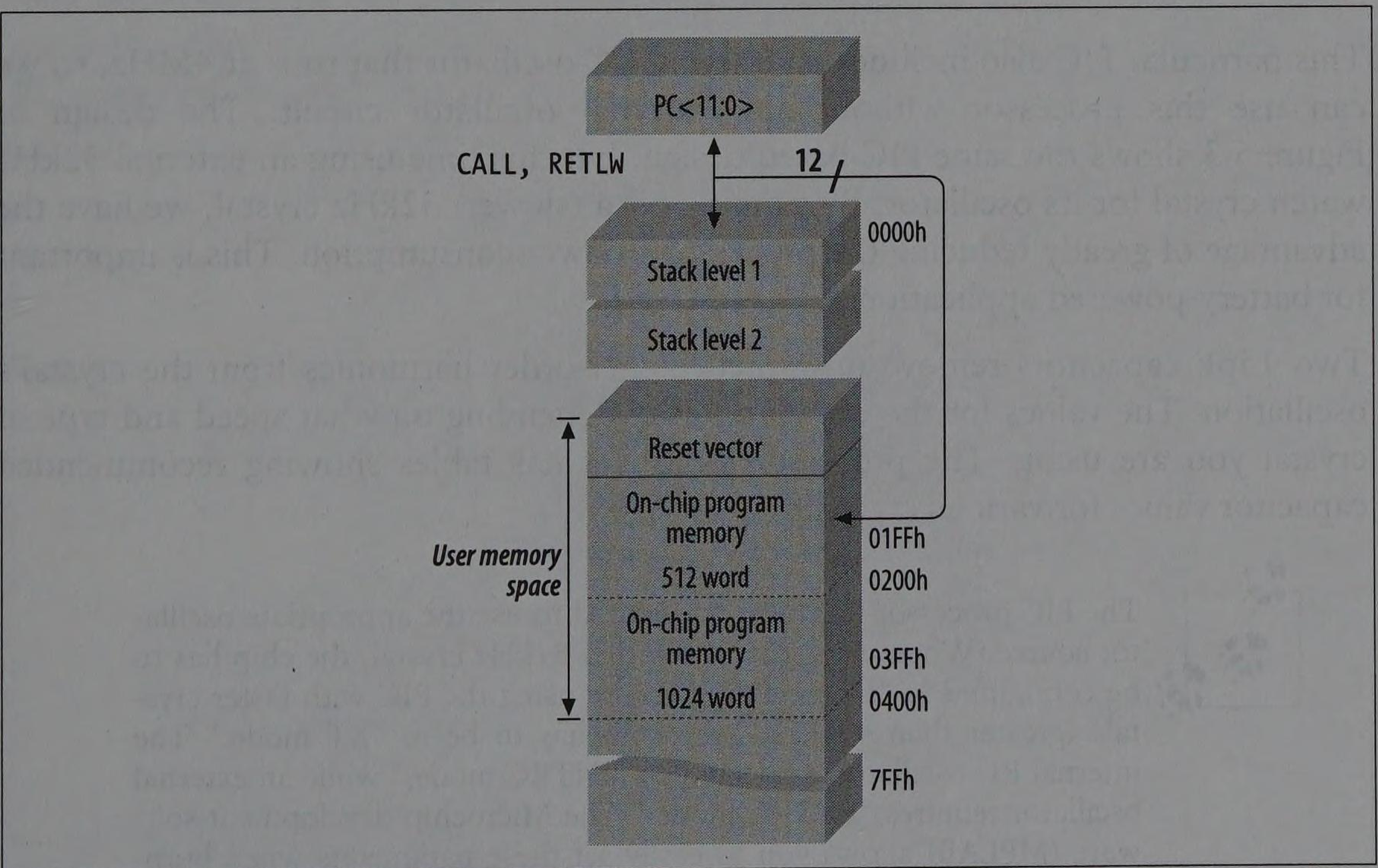


Figure 5-1. PIC12C508 program address space (Reference: PIC12C508 datasheet)

The PIC12C508 has 512 words of internal program memory and just 25 bytes of internal RAM.

Figure 5-2 shows the schematic for a small computer based upon the PIC12C508. The digital I/O signals of the PIC are brought out through a 7-pin connector. If the design were implemented using surface-mount components wherever possible, the connector would be the largest component on the PCB!

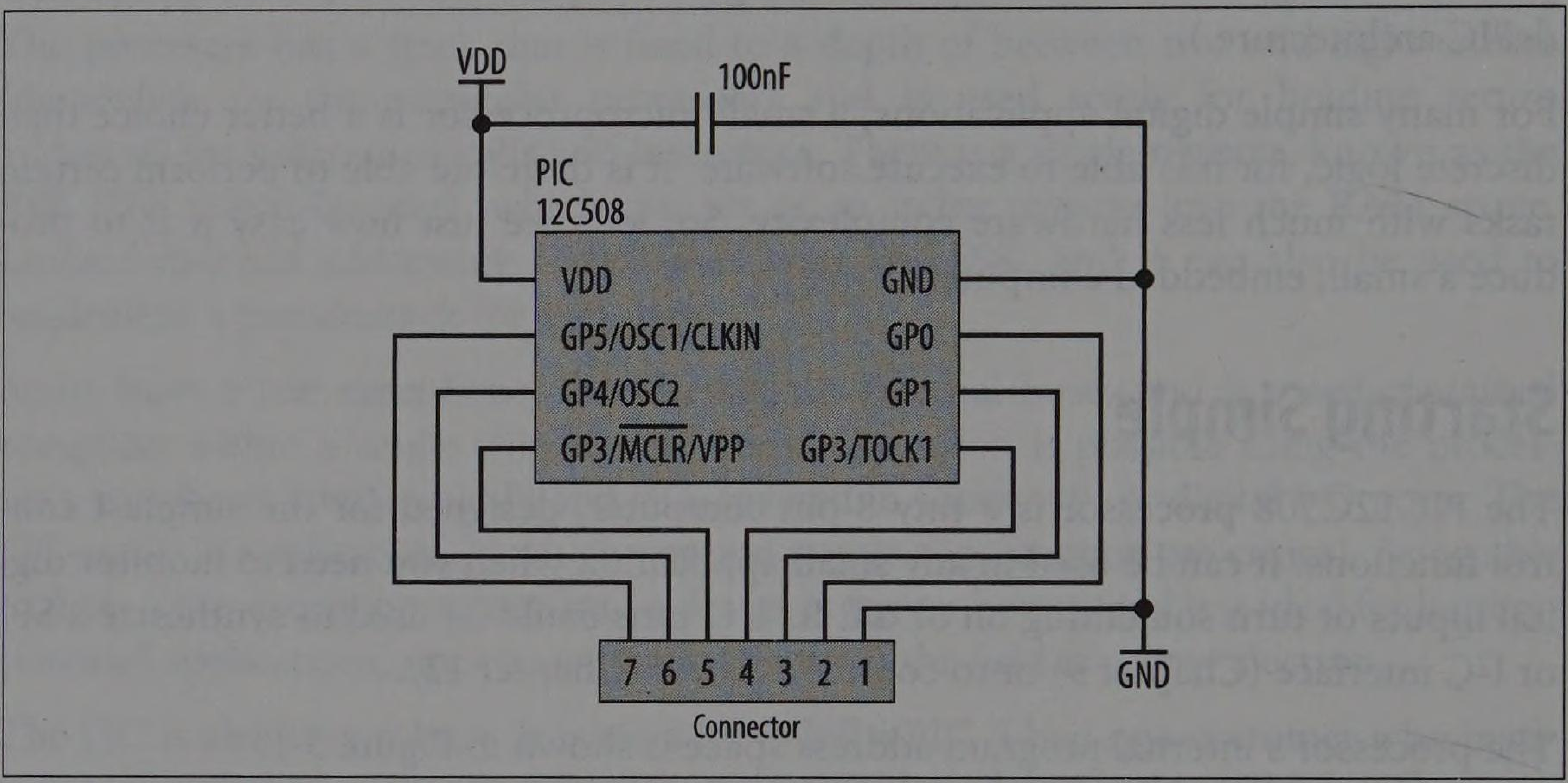


Figure 5-2. Minimal PIC12C508 computer

This particular PIC also includes an internal RC oscillator that runs at 4MHz, so we can use this processor without any external oscillator circuit. The design in Figure 5-3 shows the same PIC-based design, but this time using an external 32kHz watch crystal for its oscillator. By running off a (slower) 32kHz crystal, we have the advantage of greatly reducing the processor's power consumption. This is important for battery-powered applications.

Two 15pF capacitors remove unwanted higher-order harmonics from the crystal's oscillation. The values for the capacitors vary depending on what speed and type of crystal you are using. The processor datasheet has tables showing recommended capacitor values for various crystal frequencies.

> The PIC processor has to be configured to use the appropriate oscillator source. When using the PIC with a 32kHz crystal, the chip has to be configured in "LP mode." If you're using the PIC with faster crystals (greater than 455kHz), the chip has to be in "XT mode." The internal RC oscillator is selected by "INTRC mode," while an external oscillator requires "EXTRC mode." The Microchip development software (MPLAB) allows you to easily set these parameters when burning software into the processor.

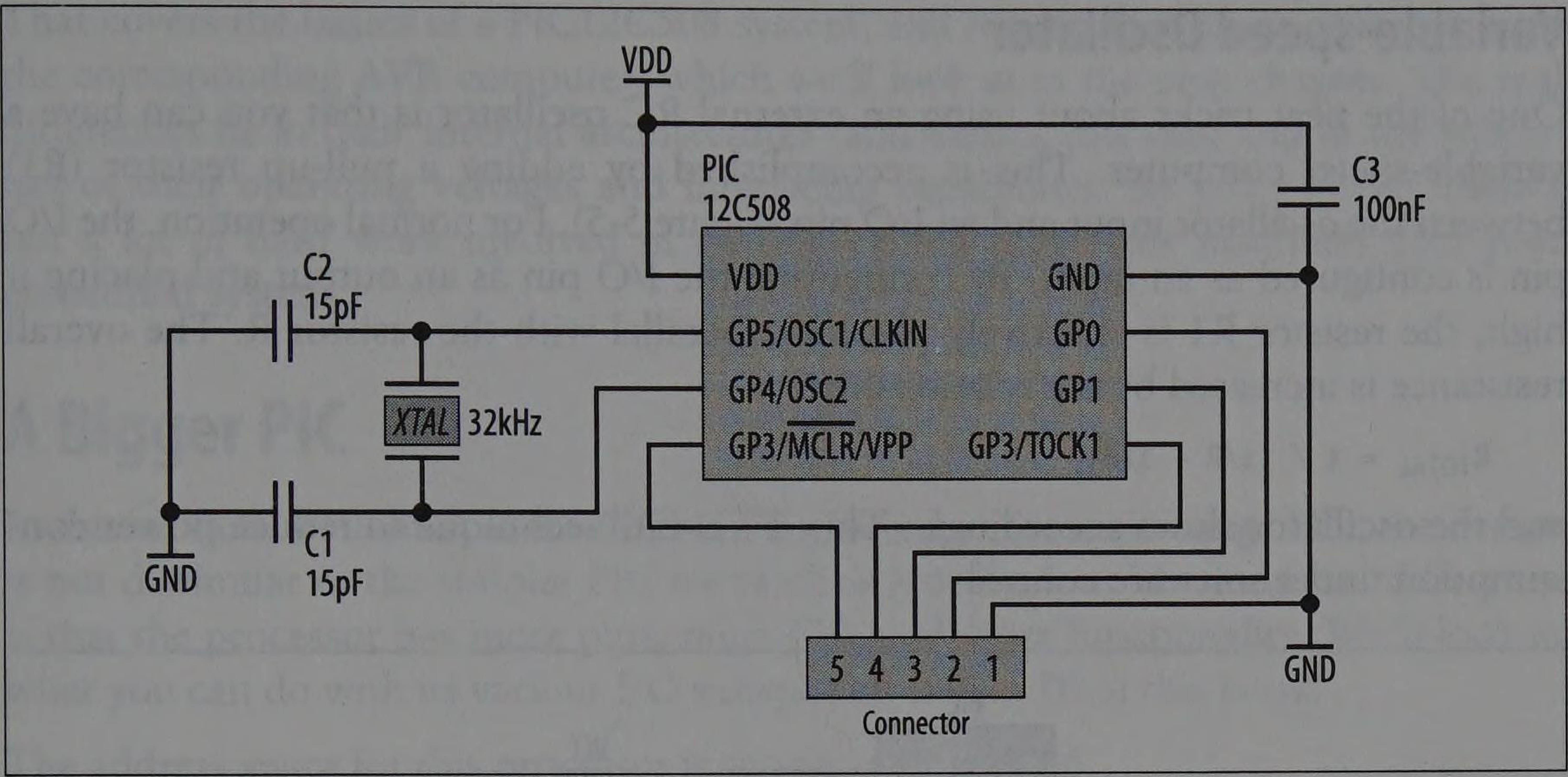


Figure 5-3. A basic PIC12C508 computer; just add power

The alternative clock source is an external RC circuit (Figure 5-4). While not the most precise timing option, it is by far the cheapest. The actual frequency of oscillation depends on a combination of the values of the resistor, the capacitor, the supply voltage, the variation in tolerances for the components, and the current operating temperature. To be clear, only an approximate operating frequency can be determined for an RC oscillator. For stable operation, Microchip recommends that the resistor should be between 3kΩ and 100kΩ, and the capacitor greater than 20pF. If you wish to use an external RC oscillator, refer to the processor's datasheet, as Microchip has detailed information on RC component selection, taking into account voltage and temperature effects.

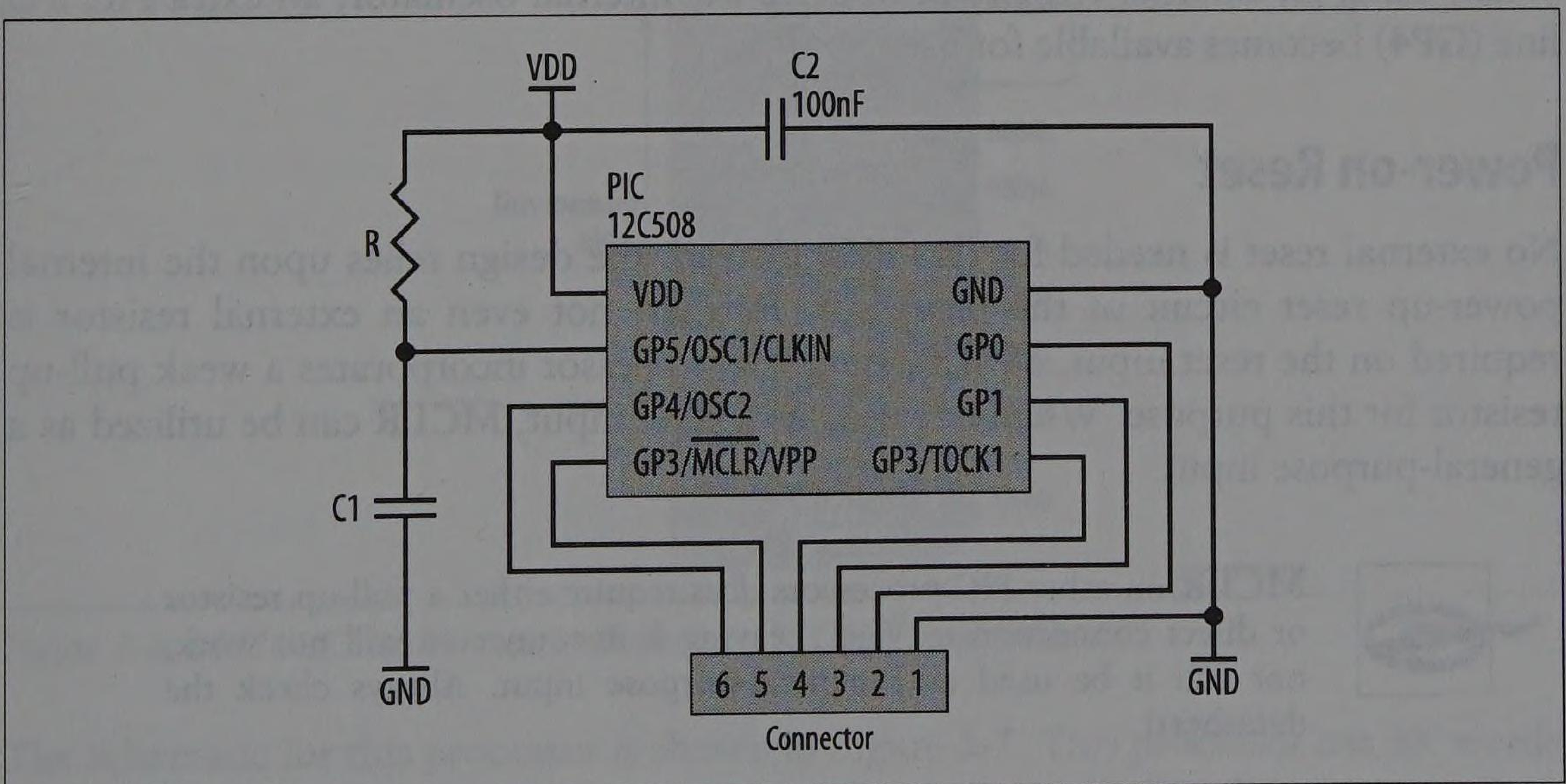


Figure 5-4. External RC oscillator

Variable-speed Oscillator

One of the neat tricks about using an external RC oscillator is that you can have a variable-speed computer. This is accomplished by adding a pull-up resistor (R1) between the oscillator input and an I/O pin (Figure 5-5). For normal operation, the I/O pin is configured as an input. By configuring the I/O pin as an output and placing it high, the resistor R1 is effectively placed in parallel with the resistor R. The overall resistance is increased by the relationship:

$$R_{TOTAL} = 1 / (1/R + 1/R1)$$

and the oscillator slows accordingly. This is a useful technique to reduce power consumption under software control.

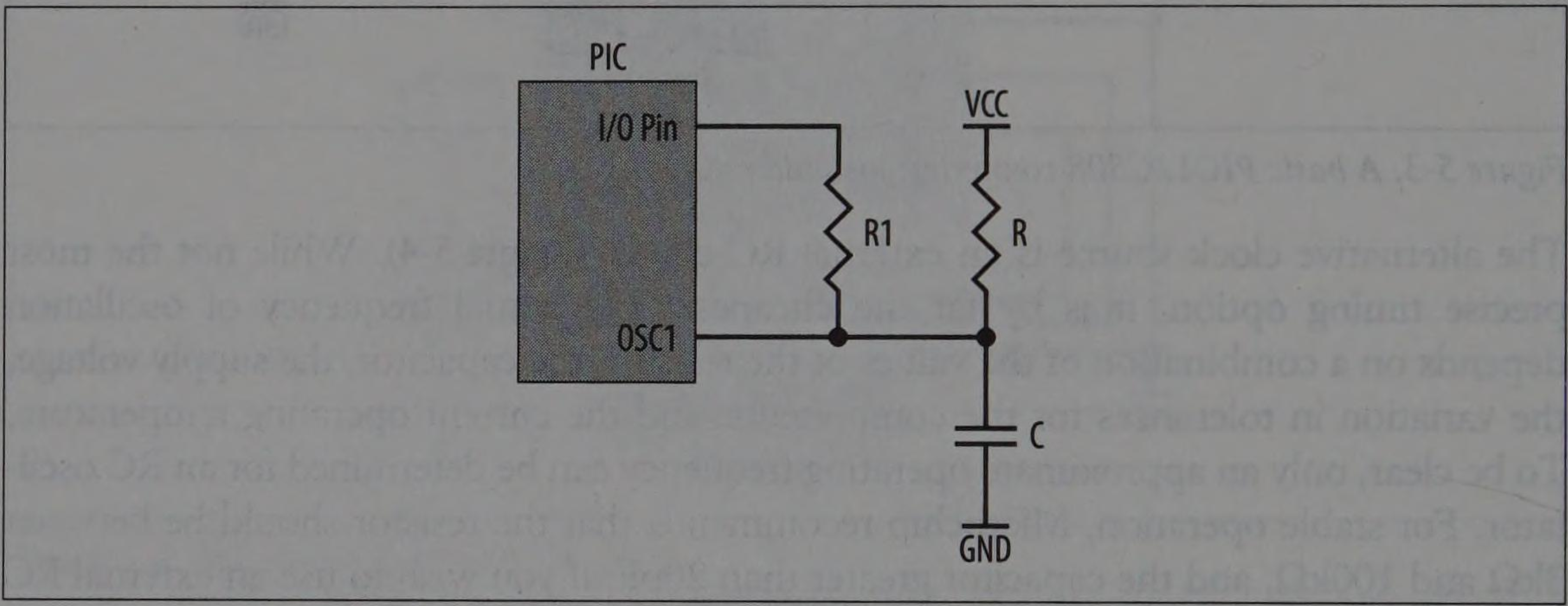


Figure 5-5. Variable-speed RC oscillator

When using an external RC circuit to drive the internal oscillator, an extra PIC I/O line (**GP4**) becomes available for use.

Power-on Reset

No external reset is needed for this PIC. Instead, the design relies upon the internal power-up reset circuit of the processor. Further, not even an external resistor is required on the reset input, **$\overline{MCLR}$**, since the processor incorporates a weak pull-up resistor for this purpose. When not used as a reset input, **$\overline{MCLR}$** can be utilized as a general-purpose input.

> $\overline{MCLR}$ on other PIC processors *does* require either a pull-up resistor or direct connection to V_{DD}. Leaving it unconnected will not work, nor can it be used as a general-purpose input. Always check the datasheet!

The power supply (**V_{DD}**) for the PIC12C508 can range from 2.5V to 5.5V.

That covers the basics of a PIC12C508 system, and it's not that much different from the corresponding AVR computer, which we'll look at in the next chapter. The real differences lie in their internal architectures (and instruction sets) and in the subtleties of their operating voltages and interfacing capabilities. As you can see, there's not a lot of hard work involved in putting one of these little machines into your embedded system.

A Bigger PIC

In this section, we'll look at the PIC16C73 processor. For a midrange PIC, the design is not dissimilar to the simpler PIC we've already looked at. The only real difference is that the processor has more pins, more I/O, and more functionality. We'll look at what you can do with its various I/O subsystems in Part III of this book.

The address space for this processor is shown in Figure 5-6.

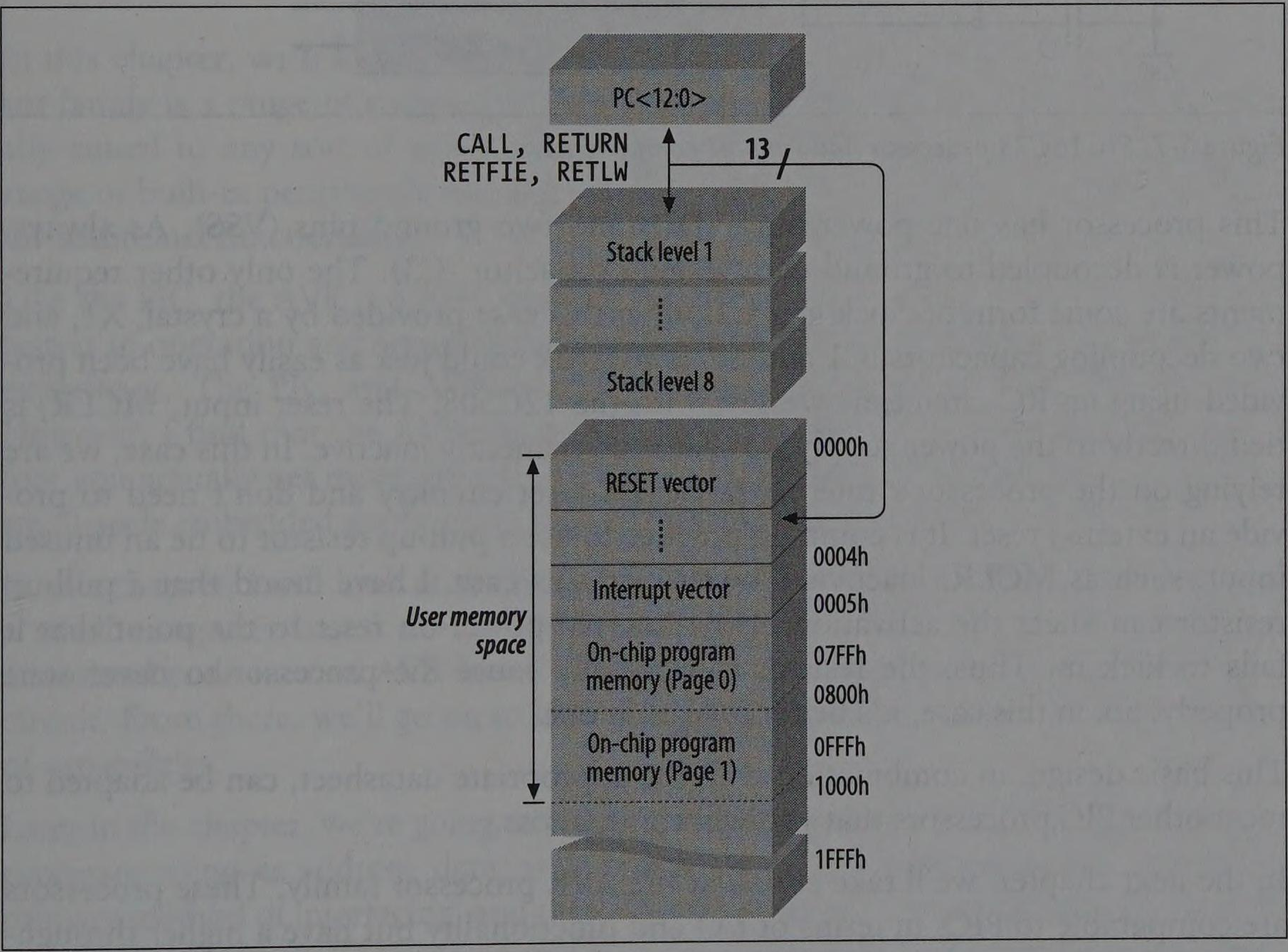


Figure 5-6. PIC16C73 address space (Reference: PIC16C73 datasheet)

The schematic for this processor is shown in Figure 5-7. This processor has 4K words of program memory, 192 bytes of RAM, and a variety of I/O subsystems, such as

three timer modules, SPI, I2C, a UART, five channels of analog input, and up to 22 digital I/O pins.

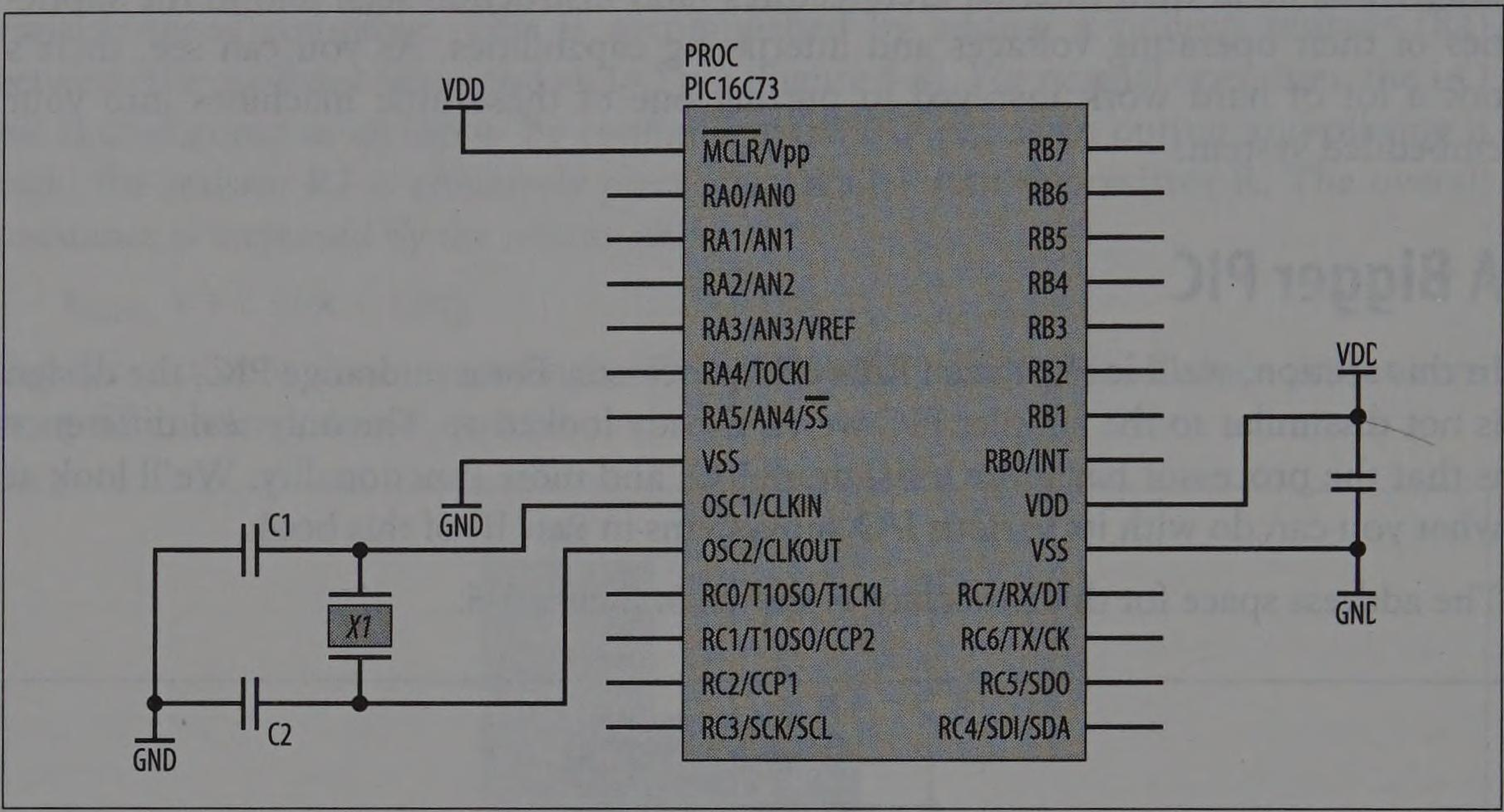


Figure 5-7. PIC16C73 processor and support components

This processor has one power pin (**VDD**) and two ground pins (**VSS**). As always, power is decoupled to ground with a small capacitor (C3). The only other requirements are some form of clock generation, in this case provided by a crystal, X1, and two decoupling capacitors, C1 and C2. The clock could just as easily have been provided using an RC circuit, as we saw with the 12C508. The reset input, **$\overline{\text{MCLR}}$**, is tied directly to the power supply, so that is permanently inactive. In this case, we are relying on the processor's internal power-on reset circuitry and don't need to provide an external reset. It is common practice to use a pull-up resistor to tie an unused input, such as **$\overline{\text{MCLR}}$**, inactive. However, in this case, I have found that a pull-up resistor can affect the activation of the internal power-on reset to the point that it fails to kick in. Thus, the resistor can actually cause the processor to never start properly. So, in this case, it's better to leave it out.

This basic design, in combination with the appropriate datasheet, can be adapted to most other PIC processors that you will come across.

In the next chapter, we'll take a look at the AVR processor family. These processors are comparable to PICs in terms of I/O and functionality but have a higher throughput and a more versatile architecture.

CHAPTER 6
The AVR Microcontrollers

A really useful engine . . .
—W. V. Awdrey

In this chapter, we'll look at the ATMEL AVR processor. Like the PIC, this processor family is a range of completely self-contained computers on chips. They are ideally suited to any sort of small control or monitoring application. They include a range of built-in peripherals and also have the capability of being expanded off-chip for additional functionality.

Like the PIC, the AVR is a RISC processor. Of the two architectures, the AVR is the fastest in operation and arguably the easiest for which to write code, in my personal experience. The PIC and AVR both approach single-cycle instruction execution. However, I find that the AVR has a more versatile internal architecture, and therefore you actually get more throughput with it. If I were looking for a processor for a small-scale embedded application, the AVR would be my first port of call.

In this chapter, I will look at the basics of creating computer hardware by designing a small computer based on the AVR, the ATtiny15. We'll also see how you can download code into an AVR-based computer and how it can be reprogrammed in-circuit. From there, we'll go on to look at some larger AVR processors, with a range of capabilities.

Later in the chapter, we're going to look at interfacing memory (and peripherals) to a processor using its address, data, and control buses. For most processors, this is the primary method of interfacing, and therefore the range of memory devices and peripherals available is enormous. You name it, it's available with a bus interface. So, knowing how to interface bus-based devices opens up a vast range of possibilities for your embedded computer. You can add RAM, ROM (or flash), serial controllers, parallel ports, disk controllers, audio chips, network interfaces, and a host of other devices.

Most small microcontrollers are completely self-contained and do not "bring out" the buses to the external world. In this chapter, we'll take a look at the ATMEL

AT90S8515 processor. It is the only processor of the AVR family that allows you access to the CPU's buses. But first, let's take a look at the AVR architecture in general.

The AVR Architecture

The AVR, developed in Norway, is produced by the ATMEL Corporation. It is a Harvard-architecture RISC processor designed for fast execution and low power consumption. It has 32 general-purpose 8-bit registers (`r0` to `r31`), six of which can also act as three 16-bit index registers (`X`, `Y`, and `Z`) (Figure 6-1). With 118 instructions, it has a versatile programming environment.

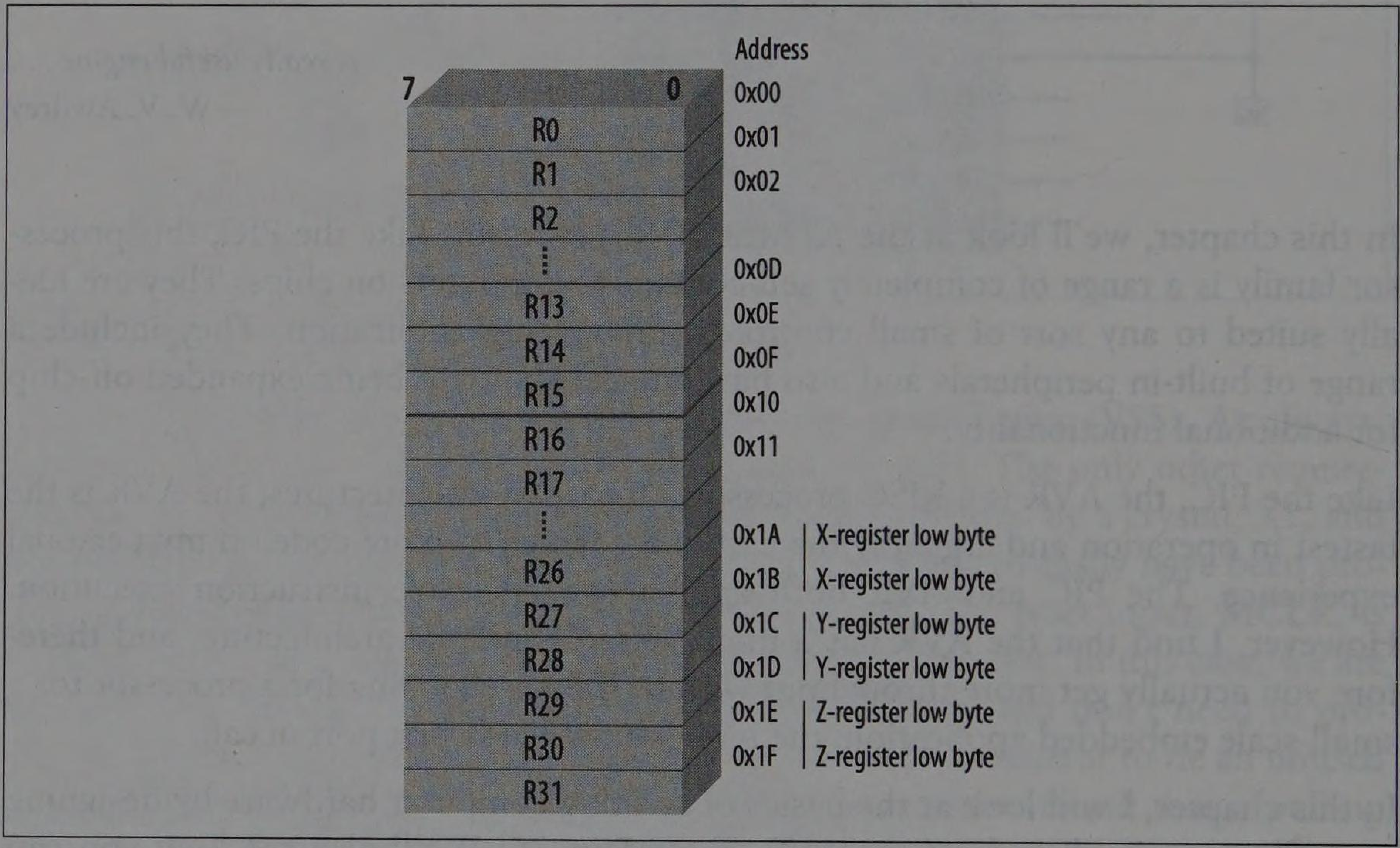


Figure 6-1. AVR registers

In most AVRs, the stack exists in the general memory space. It may therefore be manipulated by instructions and is not limited in size as is the PIC's stack.

The AVR has separate program and data spaces and supports an address space of up to 8M. As an example, the memory map for an AT90S8515 AVR processor is shown in Figure 6-2.

ATMEL is very proud of the throughput of the AVR. The company gives the following sample C code, which it compiled and ran on several different processors:

```
int max(int *array)
{
    char a;
    int maximum = -32768;
```

```
    for (a = 0; a < 16; a++)
      if (array[a] > maximum)
        maximum = array[a];
    return (maximum);
  }
```

Their results are interesting (Table 6-1).

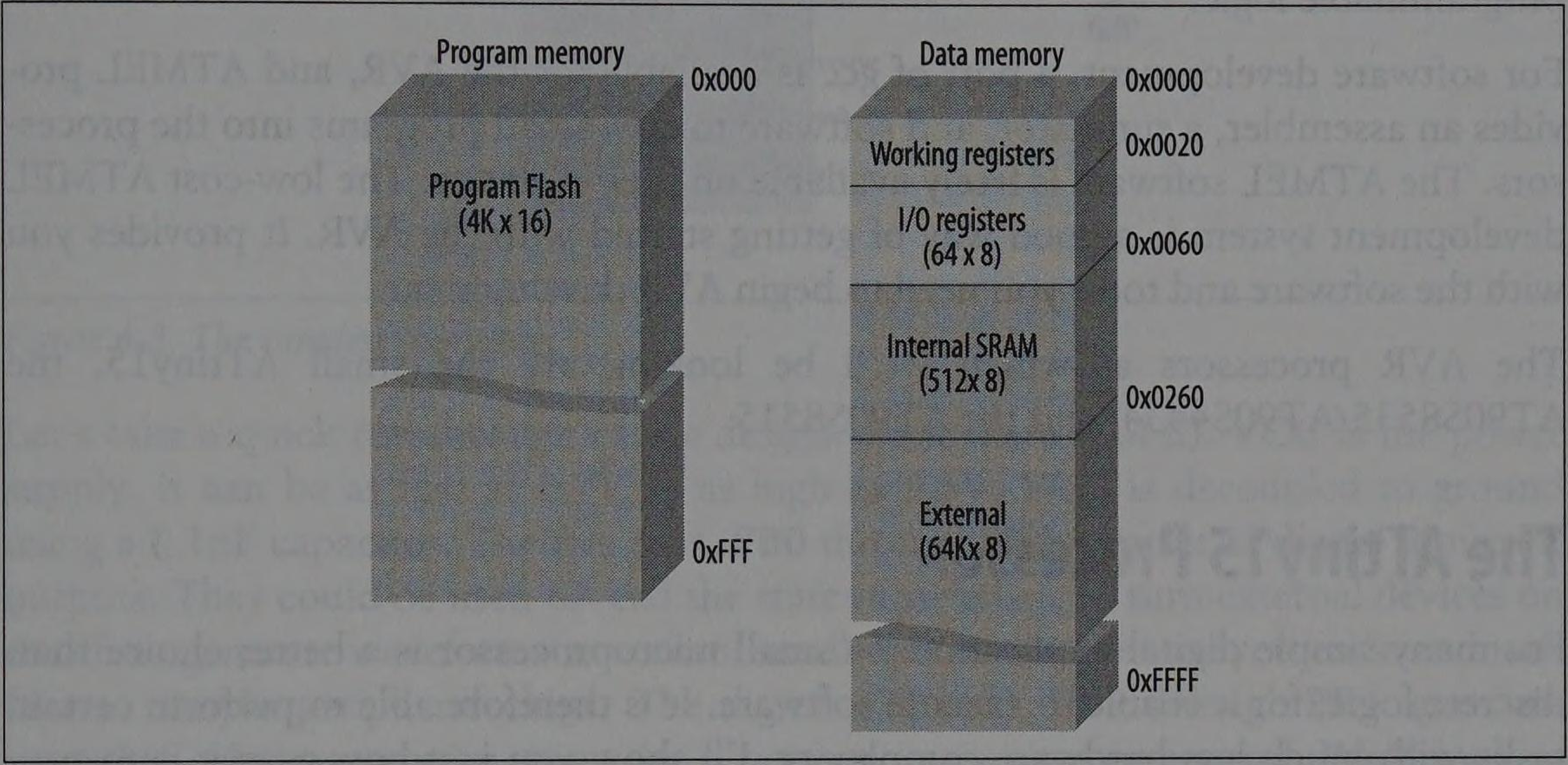


Figure 6-2. ATMEL AT90S8515 memory map

Table 6-1. ATMEL's comparison of processor speed and efficiency

Processor	Compiled code size	Execution time (cycles)
AVR	46	335
8051	112	9,384
PIC16C74	87	2,492
68HC11	57	5,244

This indicates that, when running at the same clock speed, an AVR is 7 times faster than a PIC16, 15 times faster than a 68HC11, and a whopping 28 times faster than an 8051. Alternatively, you'd have to have an 8051 running at 224MHz to match the speed of an 8MHz AVR. Now, ATMEL doesn't give specifics of which compiler(s) it used for the tests, and results can certainly be tweaked one way or the other with appropriately chosen source code. However, my personal experience is that, with the AVR, you certainly do get significantly denser code and much faster execution. For most small-scale applications, the AVR is my first choice, and it is the processor architecture I will be concentrating on in this chapter. That the AVR is faster than a corresponding PIC may change with the introduction of the new dsPIC processor by

Microchip, scheduled for release late in 2002. The dsPIC is an impressive architecture and should prove an extremely capable processor.

There are three basic families within the AVR architecture. The original family is the AT90xxxx. For complex applications, there is the ATmega family, and for small-scale use, there's the ATtiny family. ATMEL also produce large *FPGAs (Field-Programmable Gate Arrays)*, which incorporate an AVR core along with many thousands of gates of programmable logic.

For software development, a port of gcc is available for the AVR, and ATMEL provides an assembler, a simulator, and software to download programs into the processors. The ATMEL software is freely available on their web site. The low-cost ATMEL development system is a good way of getting started with the AVR. It provides you with the software and tools you need to begin AVR development.

The AVR processors at which we'll be looking are the small ATtiny15, the AT90S8535/AT90S4434, and the AT90S8515.

The ATtiny15 Processor

For many simple digital applications, a small microprocessor is a better choice than discrete logic, for it is able to execute software. It is therefore able to perform certain tasks with much less hardware complexity. I'll show you just how easy it is to produce a small, embedded computer for integration into a larger system, using an ATMEL ATtiny15 AVR processor. This processor has 512 words of flash for program storage and no RAM! (Think on that when next you have to install some 100-megabyte application on your desktop computer!) This tiny processor, unlike its bigger AVR siblings, relies solely on its 32 registers for working-variable storage.

Since there is no RAM in which to allocate stack space, the ATtiny15 instead uses a dedicated hardware stack that is a mere *three* entries deep, and this is shared by subroutine calls and interrupts. (That fourth nested function call is a killer!) The program counter is 9-bits wide (addressing 512 words of program space); therefore, the stack is also 9-bits wide. Also unlike the bigger AVRs, only two of the registers (`r30` and `r31`) may be coupled as a 16-bit index register (called `Z`).

The processor also has 64 bytes of EEPROM (for holding system parameters), up to five general-purpose I/O pins, eight internal and external interrupt sources, two 8-bit timer/counters, a four-channel 10-bit analog-to-digital converter, and an analog comparator and is able to be reprogrammed in-circuit. It comes in a tiny 8-pin package, out of which you can get up to 8 MIPS performance. We're not going to worry about most of its features for the time being. That'll all be covered in later chapters when we take a look at I/O. Instead, we're just going to concentrate on how you use one for simple digital control.

Using a small microcontroller such as the ATtiny15 is very easy. The basic processor needs very little external support for its own operation. Figure 6-3 shows just how simple it is.

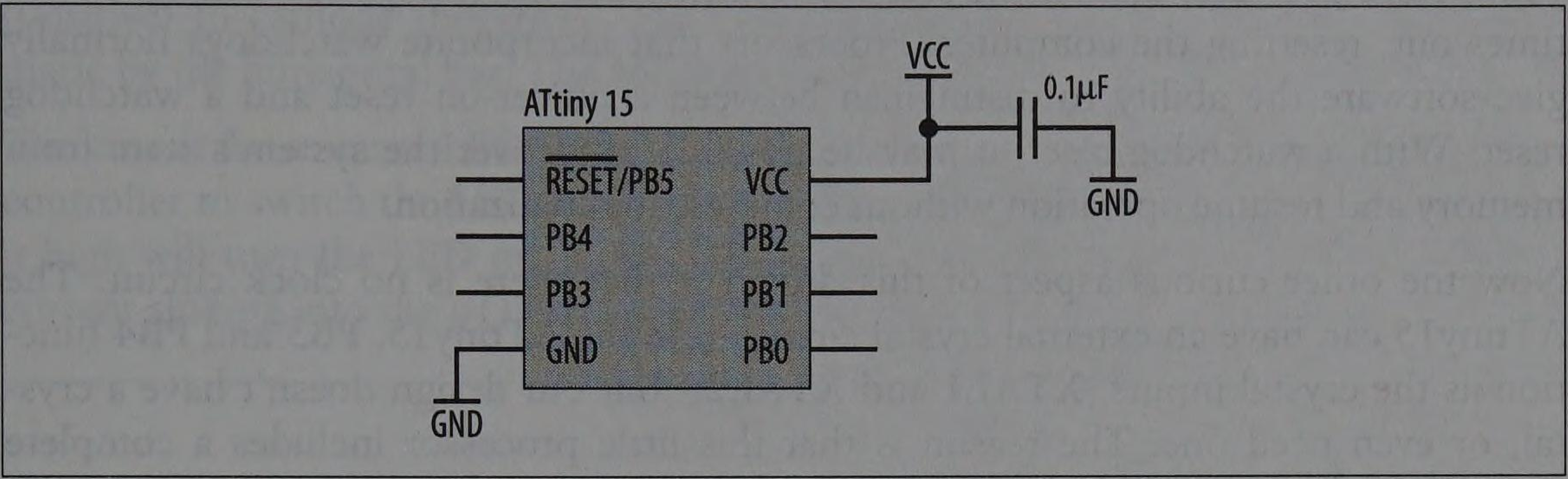


Figure 6-3. The simplest computer

Let's take a quick run-through of the design (what there is of it). **VCC** is the power supply. It can be as low as 2.7V or as high as 5.5V. **VCC** is decoupled to ground using a 0.1µF capacitor. The five pins, **PB0** through **PB4**, can act as digital inputs or outputs. They could be used to read the state of switches, to turn external devices on or off, to generate waveforms to control small motors, or even to synthesize an interface to simple peripheral chips. The digital I/O lines, **PB0** through **PB4**, get connected to whatever you're using the processor to monitor or control. We'll look at some examples of that later in the chapter.

Finally, one input, **RESET**, is left unconnected. On just about any other processor, this would be fatal. Many processors require an external *power-on reset (POR)* circuit to bring them to a known state and to commence the execution of software. Some processors have an internal power-on reset circuit and require no external support. Such processors still have a reset input, allowing them to be manually reset by a user or external system. Normally, the reset input still requires a pull-up resistor to hold it inactive. But the ATtiny15 processor doesn't require this. It has an internal power-on reset and an internal pull-up resistor. So, unlike most (maybe all) other processors, **$\overline{\text{RESET}}$** on the ATtiny15 may be left unconnected. In fact, on this particular processor, the **RESET** pin may be utilized as a general-purpose input (**PB5**) when an external reset circuit is not required. One important point: the normal input protection against higher than normal voltage inputs is not present on **$\overline{\text{RESET}}$/PB5**, since it may be raised to +12V during software download by the program burner. Therefore, you must take great care if using **PB5** that the input never exceeds VCC by more than 1V. Failing to do so may place the processor into software-download mode, and thereby effectively crash your embedded computer.

The AVR processors (and PICs too) include an internal circuit known as a *brownout detector* (*BOD*). This detects minor fluctuations on the processor's power supply that may corrupt its operation, and if such a fluctuation is detected, it generates a reset and restarts the processor. There is also an additional reset generator, known as a

watchdog, used to restart the computer in case of a software crash. It is a small timer whose purpose is to automatically reset the processor once it times out. Under normal operation, the software regularly restarts the watchdog. It's a case of "I'll reset you, before you reset me." If the software crashes, the watchdog isn't cleared and so times out, resetting the computer. Processors that incorporate watchdogs normally give software the ability to distinguish between a power-on reset and a watchdog reset. With a watchdog reset, it may be possible to recover the system's state from memory and resume operation without complete reinitialization.

Now the other curious aspect of this design is that there is no clock circuit. The ATtiny15 can have an external crystal circuit. (On the ATtiny15, **PB3** and **PB4** function as the crystal inputs, **XTAL1** and **XTAL2**.) But our design doesn't have a crystal, or even need one. The reason is that this little processor includes a complete internal oscillator (in this case, an RC oscillator), running at a frequency of 1.6MHz, and so requires no external components for its clock. The catch is that RC oscillators are not that stable and have the tendency to vary their frequency as the temperature changes. (The ATtiny15's oscillator can vary between 800kHz and 1.6MHz.) Generally, an RC oscillator is not really suitable for timing-critical applications (in which case, you'd use an external crystal instead). But if your ATtiny15 is just doing simple control functions, timing may not be an issue. You can therefore get by with using the internal RC oscillator and save on complexity. ATMEL provides an 8-bit calibration register (OSCCAL) in the ATtiny15 that enables you to tune the internal oscillator, thus making it more accurate.

There we have the basic design for an ATtiny15 machine. In essence, it's a very cheap, small, and versatile computer that requires no work for the core design. The only design effort needed is to ensure that the computer will work correctly with the I/O devices to which it is interfaced. If you're going to power the system off a battery, then the capacitor is optional as well! The only component that *must* be there is the processor itself. (And you thought designing computer hardware was going to be hard.)

That's the basic AVR computer hardware, with minimal components. We'll look at how you download software to it shortly.

So, that covers the basics of a ATtiny15 system, and it's not that much different from the corresponding PIC12C508 computer. The real differences lie in their internal architectures (and instruction sets) and in the subtleties of their operating voltages and interfacing capabilities. As you can see, there's not a lot of hard work involved in putting one of these little machines into your embedded system.

So far, neither of our computers is interfaced to anything. Let's start with something simple, adding a LED to the AVR. The basic technique applies to all microcontrollers with programmable I/O lines, as well.

Adding a Status LED

LEDs (Light-Emitting Diodes) produce light when current flows through them. Being diodes, they conduct only if the current is flowing in the right direction, from anode (positive) to cathode (negative). The cathode end of a LED is denoted on a schematic by the horizontal bar. The anode is the triangle.

The circuit for a status LED is shown in Figure 6-4. It uses an I/O line of the microcontroller to switch the LED on or off. Sending it low will turn on the LED; sending it high will turn the LED off, as we'll soon see. The resistor (R) is used to limit the current sinking into the I/O line, as we shall also see shortly.

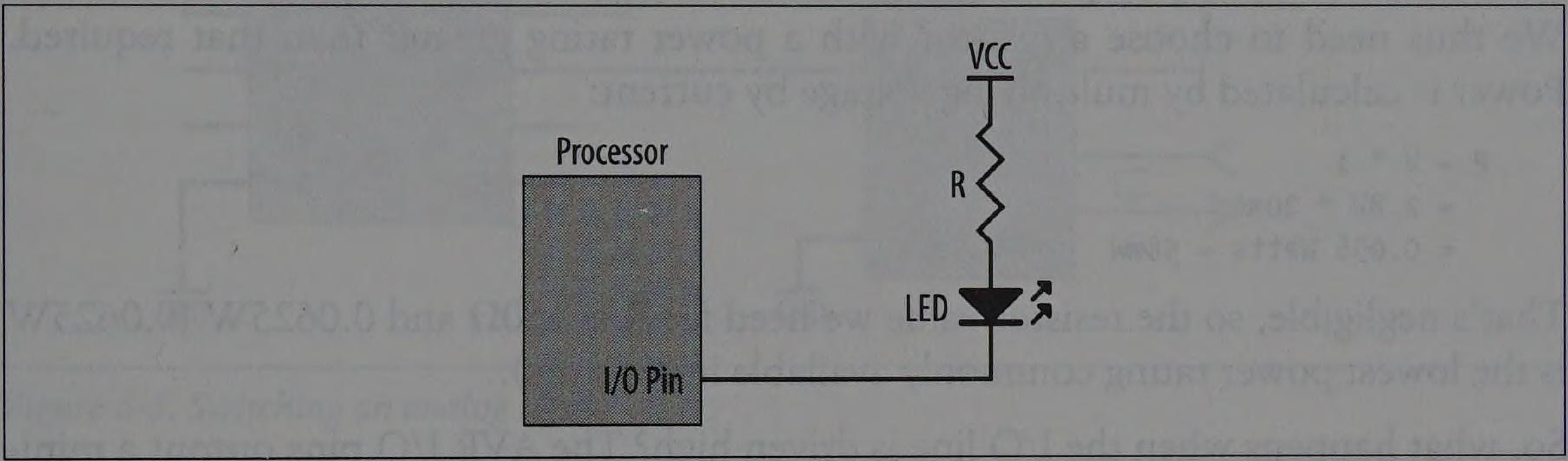


Figure 6-4. Status LED

When conducting (and thereby producing light), LEDs have a *forward voltage drop*, meaning that the voltage present at the cathode will be less than that at the anode. The magnitude of this voltage drop varies between different LED types, so check the datasheet for the particular device you are using.

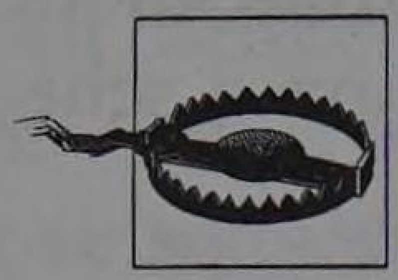
The output low voltage of an ATtiny15 I/O pin is 0.6V when the processor is operating on a 3.3V supply and 0.5V when operating on a 3V supply. Let's assume (for the sake of this example) that we are using a power supply (**VCC**) of 5V, and the LED has a forward voltage drop of 1.6V. Now, sending the output low places the LED's cathode at 0.6V. This means that the voltage difference between **VCC** (5V) and the cathode is 4.4V. If the LED has a voltage drop of 1.6V, this means that the voltage drop across the resistor is 2.8V.

```
(5V - 1.6V - 0.6V) = 2.8V
```

Now, from the datasheet, the digital I/O pins of an AVR can sink up to 20mA if the processor is running on a 5V supply. We therefore have to limit the current flow to this amount, and this is the purpose of the resistor. If the resistor has a voltage difference across it of 2.8V (as we calculated) and a current flow of 20mA, then from Ohm's Law we can calculate what value resistor we need to use:

```
R = V / I
  = 2.8V / 20mA
  = 140Ω
```

The closest available resistor value to this is 150Ω, so that's what we'll use. (That will give us an actual current of 18.6mA, which is fine.)



The AVR can sink 20mA per pin when operating on a 5V supply. However, the amount of current it can sink decreases with supply voltage. When running on a 2.7V supply, the AVR can sink only 10mA. As always, it's important to read the datasheets carefully.

The next question is: how much power will the resistor have to dissipate? In other words, how much energy will it use in dropping the voltage by 2.8V? This is important, for if we try to pump too much current through the resistor, we'll burn it out. We thus need to choose a resistor with a power rating greater than that required. Power is calculated by multiplying voltage by current:

```
P = V * I
  = 2.8V * 20mA
  = 0.056 Watts = 56mW
```

That's negligible, so the resistor value we need for R is 150Ω and 0.0625W (0.0625W is the lowest power rating commonly available in resistors).

So, what happens when the I/O line is driven high? The AVR I/O pins output a minimum of 4.3V when high (and using a 5V supply). With the output high, the voltage at the LED's cathode will be at least 4.3V, so the voltage difference between the cathode and **VCC** will be only 0.7V (or less). But, the forward voltage drop of the LED is 1.6V. Thus, there is not enough voltage across the LED to turn it on.

In this way, we can turn the LED on or off using a simple digital output of the processor. We have also seen how to calculate voltages and currents. It is very important to do this with every aspect of a design. Ignoring it can result in a nonfunctioning machine or, worse, charred components and that wafting smell of burning silicon.

We've just seen how to use the digital outputs of the AVR to control a LED. This will work with any device that uses less than 20mA. In fact, for low-power components, such as some sensors, it is possible to use the AVR's output to provide direct power control, just as we provided direct power control for the LED. In battery-powered applications, this can be a useful technique for reducing the system's overall power consumption.

Switching Analog Signals

We can also use the digital I/O lines of the processor to control the flow of analog signals within our system. For example, perhaps our embedded computer is integrated into an audio system and is used to switch between several audio sources. To do this, we use an analog switch such as the MAX4626, one for each signal path. This tiny component (about the size of a grain of rice in the surface-mount version) operates from a single supply voltage (as low as 1.8V and as high as 5.5V). It also incorporates

built-in overload protection to prevent device damage during short circuits. The schematic showing a MAX4626 interfaced to an ATtiny15 AVR is shown in Figure 6-5. Driving the AVR's output (**PB2**) high turns the MAX4626 on and makes a connection between **NO** and **COM**. Sending **PB2** low breaks the connection. In this way, the MAX4626 can be used to connect an output to an input, under software control.

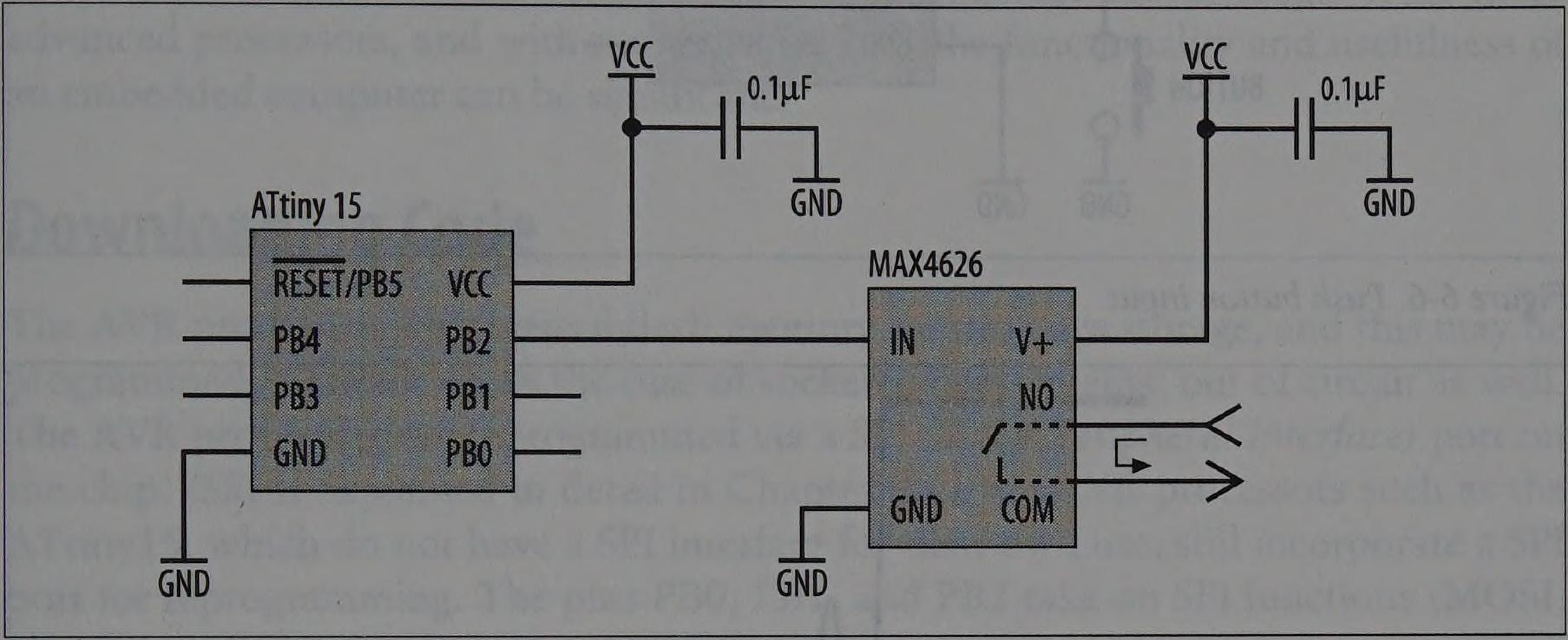


Figure 6-5. Switching an analog signal

The question is: will it work with an AVR? When operating on a 5V supply, the input to the MAX4626 (pin 4, **IN**) requires a logic low input of less than 0.8V, and a logic high input of at least 2.4V. The AVR's logic low output is 0.6V or less, and its logic high output is a minimum of 4.3V. So, the AVR's digital output voltages match the requirements of the MAX4626. As for current, the MAX4626 needs to sink or source only a minuscule 1μA. For an AVR, this is not a problem.

If the MAX4626 doesn't suit, MAXIM and other manufacturers produce a range of similar devices with varying characteristics. There's bound to be something that meets your needs.

The schematic in Figure 6-6 shows a push button connected to **PB3**, where **PB3** is acting as a digital input. Now, there are a couple of interesting things to note about this simple input circuit. The first is that there is no external pull-up resistor attached to **PB3**. Normally for such a circuit, an external pull-up resistor is required to place the input into a known state when the button is open (not being pressed). The pull-up resistor takes the input high, except when the button is closed and the input is connected directly to ground. The reason we can get away without an external pull-up resistor is that the AVR incorporates internal pull-up resistors, which may be enabled or disabled under software control.

The second interesting thing to note is that there is no debounce circuitry between the button and the input. Any sort of mechanical switch (and that includes a keyboard key) acts as a little inductor when pressed. The result is a rapid ringing oscillation on the signal line that quickly decays away (Figure 6-7).

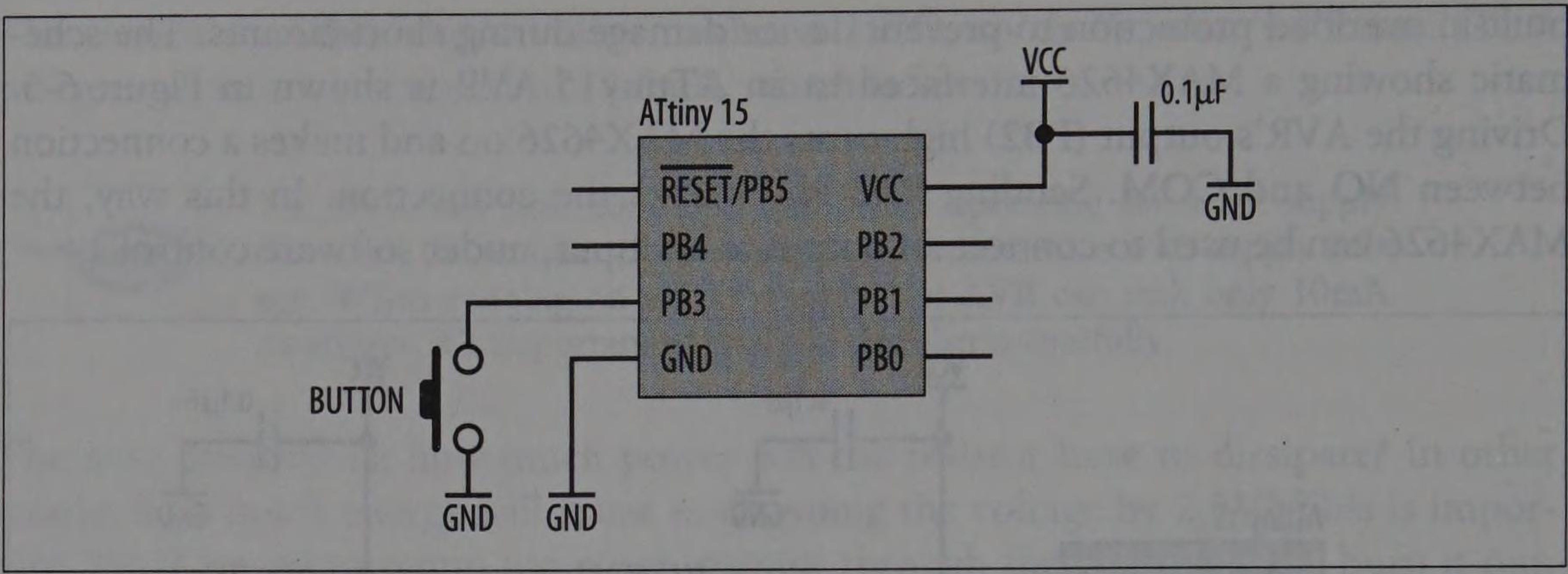


Figure 6-6. Push button input

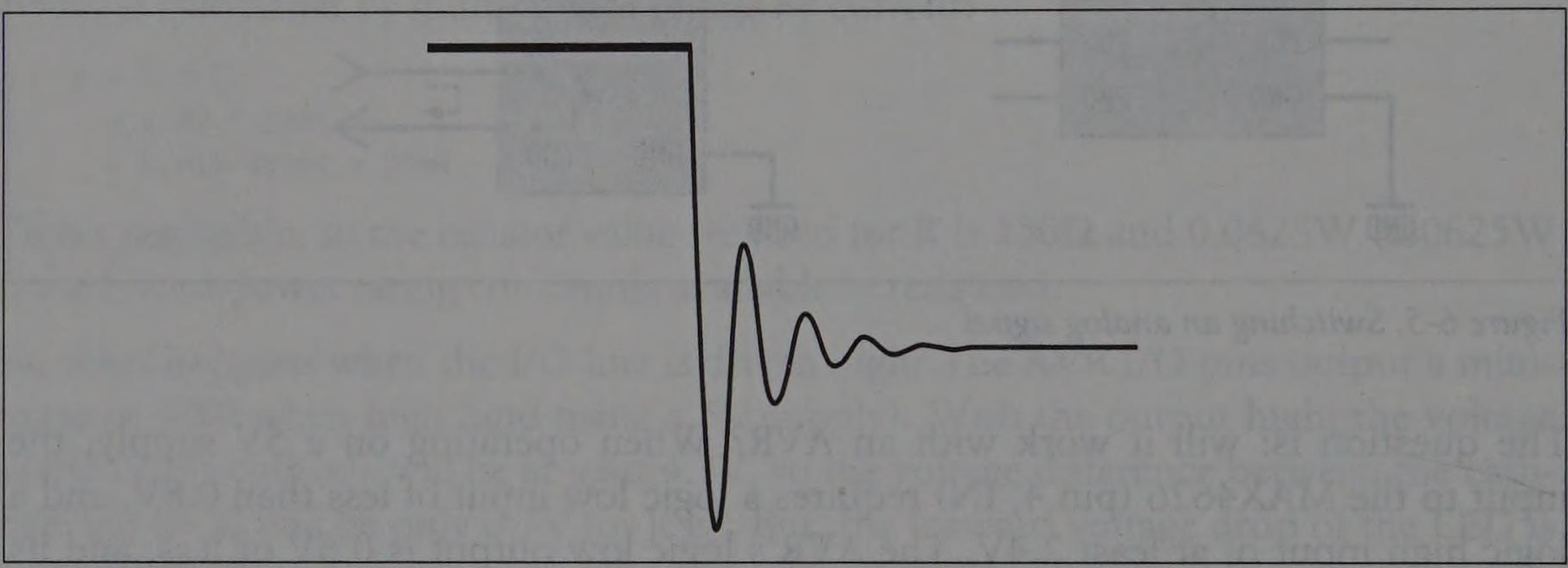


Figure 6-7. Signal bounce

So, instead of a single change of state, the resulting effect is as if the user has been rapidly hammering away on the button. Software written to respond to changes in this input will register the multiple pulses, rather than the single press the user intended. Removing these transients from the signal is therefore important and is known as *debouncing*.

Now, there are several different circuits that you could include that will cleanly remove the ringing. But here's the thing: you don't always need to! When a user presses a button, he will usually hold that button closed for at least half a second, maybe more, by which time the ringing has died away. The problem can therefore be solved in software. The software, when it first registers a low on the input, waits for a few hundred milliseconds, then samples the input again (perhaps more than once). If it is still low, then it is a valid button press, and the software responds. The software then "rearms" the input, awaiting the next press. Debouncing hardware does become important, however, if the button is connected to an interrupt line or reset.

So far, we have seen how to use the AVR to control digital outputs and read simple digital inputs. The astute among you may ask, when looking at the previous two circuits, why do we need the processor? After all, it is certainly possible to connect the button directly to the input of the MAX4626. Of what use can the processor be?

Well, we've already seen one use. The processor can replace debounce circuitry on the input. Since it has internal memory and the ability to execute software, the processor can also keep track of system state (and mode), can monitor various inputs in relation to one another, and can provide complicated control sequencing on the outputs. In short, the inclusion of a microprocessor can reduce hardware complexity while increasing system functionality. They can be very useful tools. With more advanced processors, and with more diverse I/O, the functionality and usefulness of an embedded computer can be significant.

Downloading Code

The AVR processors use internal flash memory for program storage, and this may be programmed in-circuit or, in the case of socketed components, out of circuit as well. The AVR processors are reprogrammed via a *SPI (Serial Peripheral Interface)* port on the chip. (SPI is explained in detail in Chapter 9.) Even AVR processors such as the ATtiny15, which do not have a SPI interface for their own use, still incorporate a SPI port for reprogramming. The pins **PB0**, **PB1**, and **PB2** take on SPI functions (**MOSI**, **MISO**, and **SCK**) during programming.

VCC can be supplied by the external programmer downloading the code. For programming, **VCC** *must* be 5V. If the embedded system's local supply will provide 5V, then the connection to the programmer's **VCC** may be left unmade. However, if the embedded system's supply voltage is something other than 5V, the programmer's **VCC** must be used, and any local power source within the embedded system should be disabled. **$\overline{\text{RESET}}$** plays an important role. Programming begins with **$\overline{\text{RESET}}$** being asserted (driven low). This disables the CPU within the processor and thus allows access to the internal memory. It also changes the functionality of **PB0**, **PB1**, and **PB2** to a SPI interface. The development software then sends, via the SPI interface, a sequence of codes to "unlock" the program memory and enable software to be downloaded. Once programming is enabled, sequences of write commands are performed, and the software (and other settings) are downloaded byte by byte. The ATMEL software takes care of this, so normally you don't need to worry about the specifics. If you need to do it "manually," perhaps from some other type of host computer, the ATMEL datasheets give full details of the protocol.

The ATMEL development system comes with a special adapter cable that plugs into its development board and allows you to reprogram microprocessors via a PC's parallel port. By including the right connector (with the appropriate connections) in your circuit, it's possible to use the same programming cable on your own embedded system. Depending on the particular development board, you can choose one of two possible connectors for in-circuit programming. The pinouts for these are shown in Figure 6-8. **VTG** is voltage supply for the target system. If the target has its own power source, of the appropriate voltage level for programming (+5V), then **VTG** may be left unconnected. Pin 3 is labeled as a no connect on some ATMEL application

notes; however, some development systems use this to drive a LED (indicating that a programming cycle is under way).

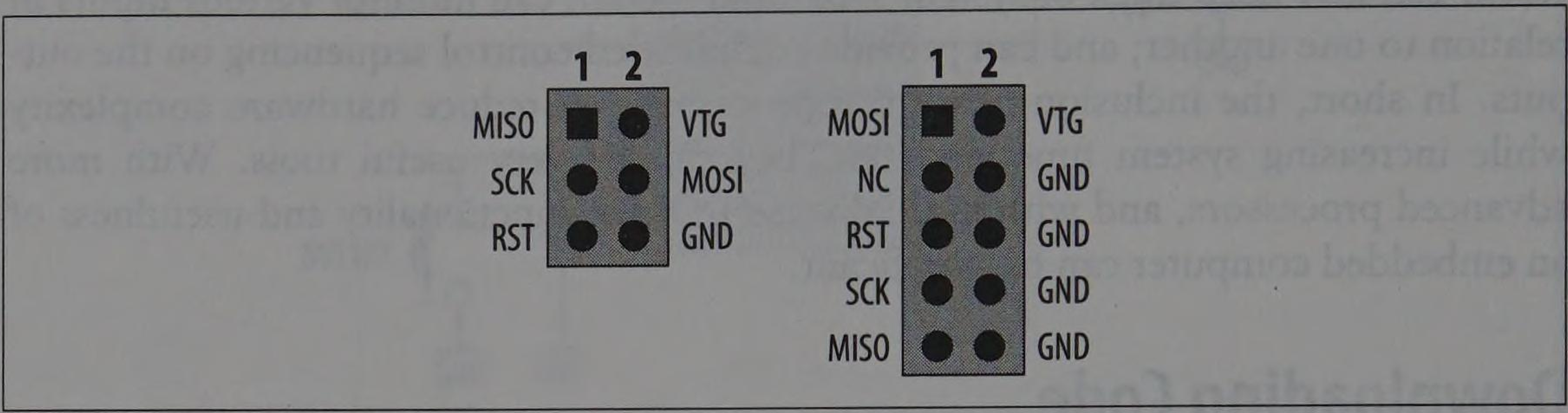


Figure 6-8. In-circuit programming connectors

The schematic to support incircuit programming is shown in Figure 6-9. Note that **MOSI** on the connector goes to **MISO** on the processor, and similarly **MISO** goes to **MOSI** on the processor. This is because, during programming, the processor is a slave and not a master.

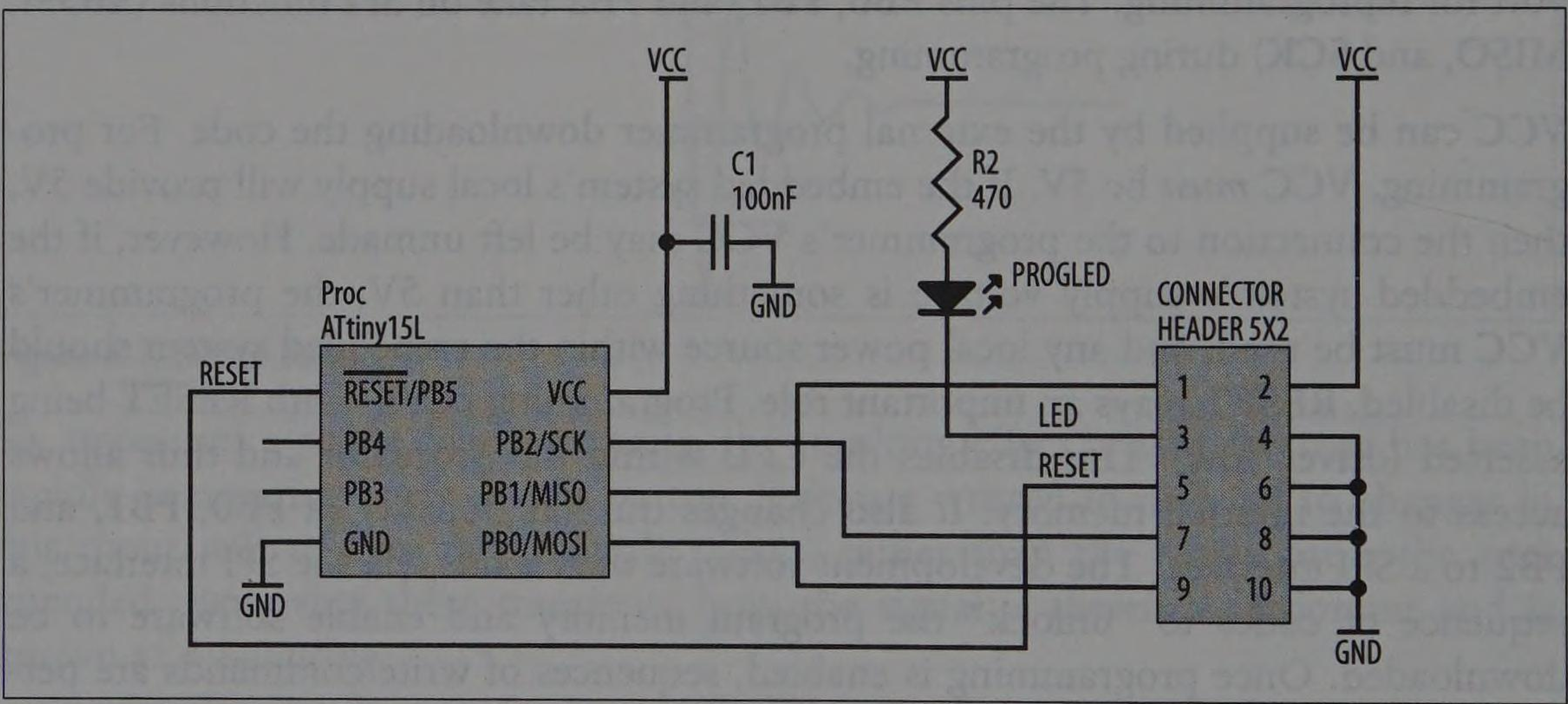


Figure 6-9. In-circuit programming

The connector type is an IDC header, and the cable provides all the signals necessary for programming, including one to drive a programming indicator LED. When not being used for programming, the connector may also double as a simple I/O connector for the embedded computer, allowing access to the digital signals. Thus, the one piece of hardware can assume dual roles.

An important note, however: if you use **PB0, PB1,** or **PB2** to interface to other components within your computer, care must be taken that the activity of programming does not adversely affect them. For example, our circuit with the MAX4626 used **PB2** as the control input. During programming, **PB2** acts as **SCK,** a clock signal. Therefore, the MAX4626 would be rapidly turned on and off as code was downloaded to the processor. If the MAX4626 was controlling something, that device

would also rapidly turn on and off, with potentially disastrous effects. Conversely, if there are other components in your system, they must not attempt to drive a signal onto **PB0**, **PB1**, or **PB2** during the programming sequence. To do so would, at the very least, result in a failed download and at the worst damage both the embedded system *and* the programmer. It's therefore vitally important to consider the implications of in-circuit programming on other components within the system.

So, what's the answer? Well, we could use **PB3** to control the MAX4626 instead, since it doesn't take part in the programming process. Alternatively, if we needed to use **PB2**, we could provide a buffer between the processor and the MAX4626, perhaps controlled by $\overline{\textbf{RESET}}$. When $\overline{\textbf{RESET}}$ is low (during programming), the buffer is disabled and the MAX4626 is isolated. Another solution may simply be to use a DIP version of the processor, mounted via a socket, and physically remove it for reprogramming. If you're using a surface-mount version of the processor, perhaps the processor could be mounted on a small PCB that plugs into the embedded computer (much like a memory SIMM on a desktop computer) and may be removed for programming. There are plenty of alternatives, and which is best really depends on your application.

Some AVRs (not the ATtiny15) have the capability of modifying their own program memory with the `SPM` (Store Program Memory) instruction. With such processors, your software can download new code via the processor's serial port and write this into the program memory. To do this, you need to have your processor preprogrammed with a bootloader program. Normally, you would load all your processors with the bootloader (and Version 1.0 of your application software) during construction. The self-programming can then be used to update the application software when the systems are out in the field. To facilitate this, the program memory is divided into two separate sections: a boot section and an application section. The memory space is divided into pages of either 128 or 256 bytes (depending on the particular processor). Memory must be erased and reprogrammed one page at a time. During programming, the Z register is used as a pointer for the page address, and the `r1` and `r0` registers together hold the data word to be programmed. The ATMEL application note (*AVR109: Self-programming*), available on the company's web site, gives example source code for the bootloader and explains the process in detail.

No matter what processor you are using, the technical data from the chip manufacturer will tell you how you go about putting your code into the processor.

A Bigger AVR

So far, we have looked at a small AVR with very limited capabilities. In Part III of this book, we will look at various forms of input and output commonly found in embedded systems. For this, we will need processors with more functionality. We have exhausted the ATtiny15 and so now need to move on to processors with a bit more

"grunt." Before getting into the detail of I/O in the later chapters, I'll introduce these processors to you and show you what you need to do to include them in your design.

The first processor is the ATMEL AT90S8535. This is a midrange AVR with lots of built-in I/O. As well as digital I/O, it has a variety of interfaces such as a serial port, SPI, analog input, timers, and counters. We'll talk about some of these interfaces in detail in later chapters, but for the moment, we'll concentrate on the processor itself.

The processor has 512 bytes of internal RAM and 8K of flash memory for program storage. Its smaller sibling, the AT90S4434, is identical in every way except that it has smaller memory spaces of 4K for program storage and 256 bytes of RAM. But from an electronics point of view, the AT90S8535 and the AT90S4434 are the same.

The basic schematic for an AT90S8535-based computer, without any extras, is shown in Figure 6-10. It is not that different from the ATtiny15, save that it has a lot more pins. **RESET** has an external 10k pull-up resistor. The processor has an external crystal (X1), and this requires two small bypass capacitors, C1 and C2. There are four power pins for this processor, and each is decoupled with a 100nF ceramic capacitor. One of the power inputs (**AVCC**) is the power supply for the analog section of the chip, and this is isolated from the digital power supply by a 100Ω resistor, R2. This is to provide a small barrier between the analog section and any switching noise that may be present from the digital circuits. The remaining pins are general-purpose digital I/O, as with the ATtiny15. However, unlike the ATtiny15, these pins have dual functionality. They may be configured, under software control, for alternative I/O functions. The processor's datasheet gives full details for configuring the functionality of the processor under software control.

That basic AVR design is applicable to most AVRs that you will find. The pinouts may be different, but the basic support required is the same. As with everything, grab the appropriate datasheet, and it will tell you the specifics for the particular processor that you are using.

Bus Interfacing

In this section, I'll show you how to expand the capabilities of your processor by interfacing it to bus-based memories and peripherals. Before we do anything else, let's take a quick tour of those mysterious timing diagrams found in datasheets and understand what they all mean.

Timing

A *timing diagram* is a representation of the input and output signals of a device and how they relate to one another. In essence, it indicates when a signal needs to be asserted and when you can expect a response from the device. For two devices to interact, the timing of signals between the two must be compatible, or you must provide additional circuitry to make them compatible.

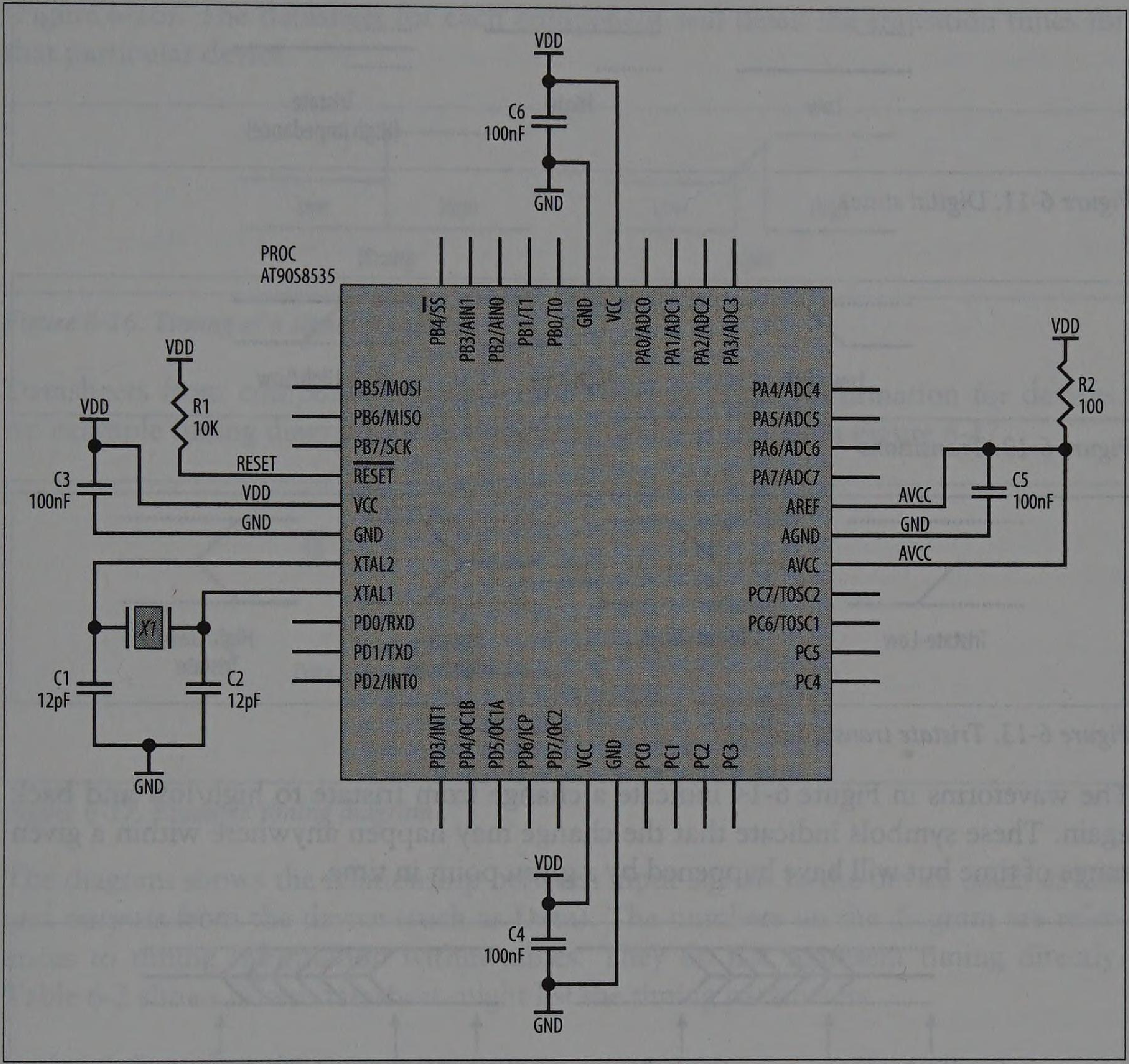


Figure 6-10. AT90S8535 processor and support components

Timing diagrams scare and confuse many people and are often ignored completely. Ignoring device timing is a sure way of guaranteeing that your system will not work! However, they are not that hard to understand and use. If you want to design and build reliable systems, remember that timing is everything!

Digital signals may be in one of three states, high, low, or high impedance (tristate). On timing diagrams for digital devices, these states are represented as shown in Figure 6-11.

Transitions from one state to another are shown in Figure 6-12.

The last waveform (High-High/Low) indicates that a signal is high and, at a given point in time, may either remain high or change to low. Similarly, a signal line that is tristate may go low, high, or either high or low depending on the state of the system. An example of this is a data line, which will be tristate until an information transfer begins. At this point in time, it will either go high (data = 1) or low (data = 0) (Figure 6-13).

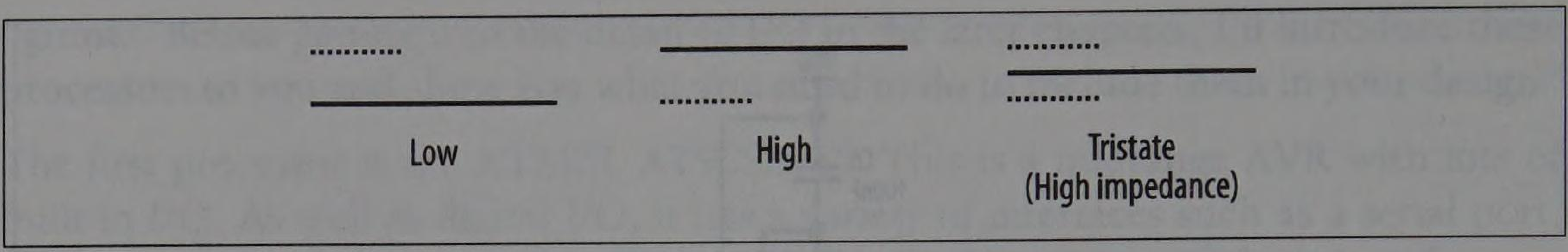


Figure 6-11. Digital states

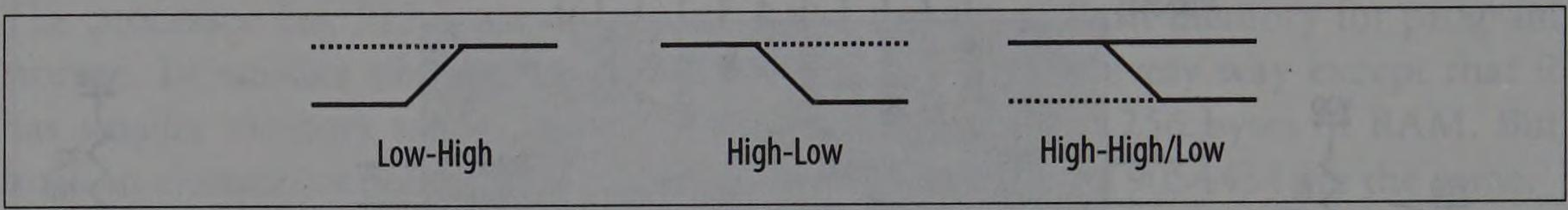


Figure 6-12. Transitions

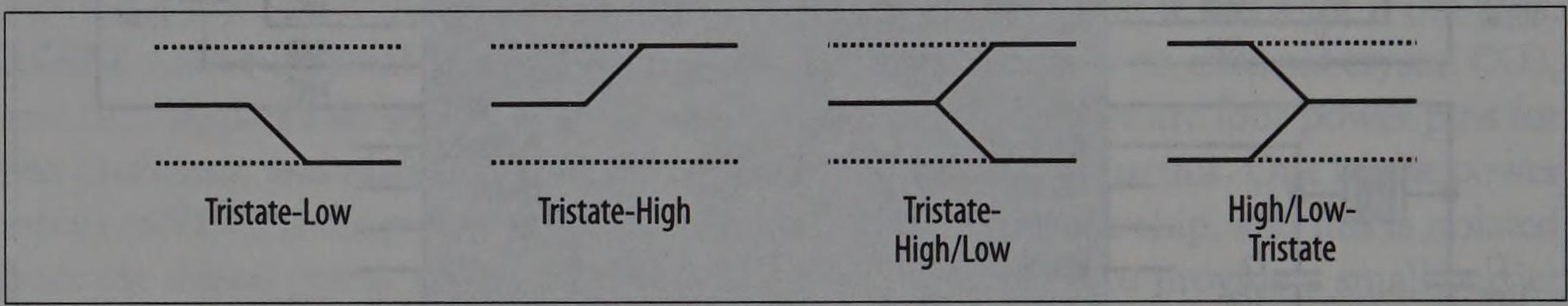


Figure 6-13. Tristate transitions

The waveforms in Figure 6-14 indicate a change from tristate to high/low and back again. These symbols indicate that the change may happen anywhere within a given range of time but will have happened by a given point in time.

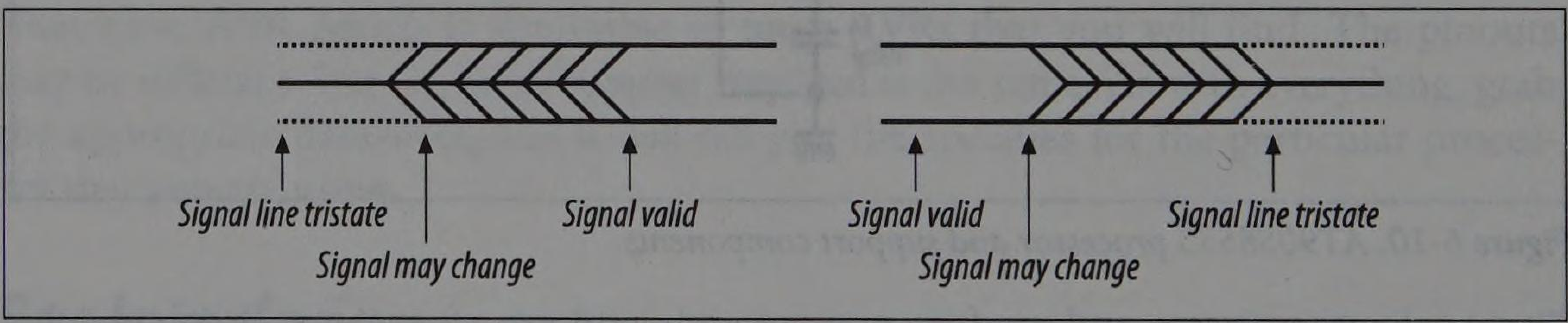


Figure 6-14. Transition timing

The waveform in Figure 6-15 indicates that a signal may/will change at a given point in time. The signal may have been high and will either remain high or go low. Alternatively, the signal may have been low and will either remain low or go high.

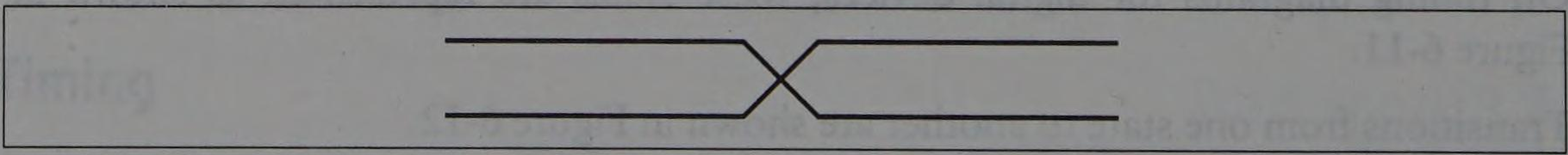


Figure 6-15. Change in signal state

The impression given in many texts on digital circuits is that a change in signal state is instantaneous. This is not so. A transition is never instant; it can be several nanoseconds in duration, and there is considerable variation between different devices

(Figure 6-16). The datasheet for each component will detail the transition times for that particular device.

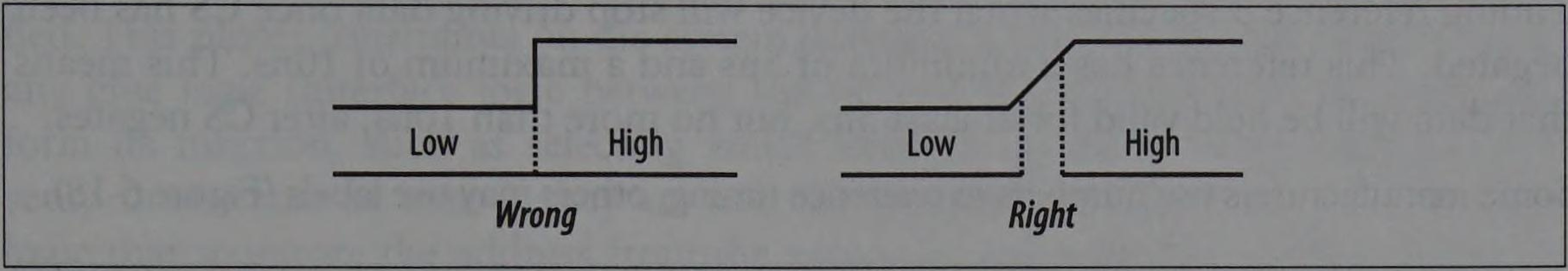


Figure 6-16. Timing of a signal transition

Datasheets from component manufacturers specify timing information for devices. An example timing diagram for an imaginary device is shown in Figure 6-17.

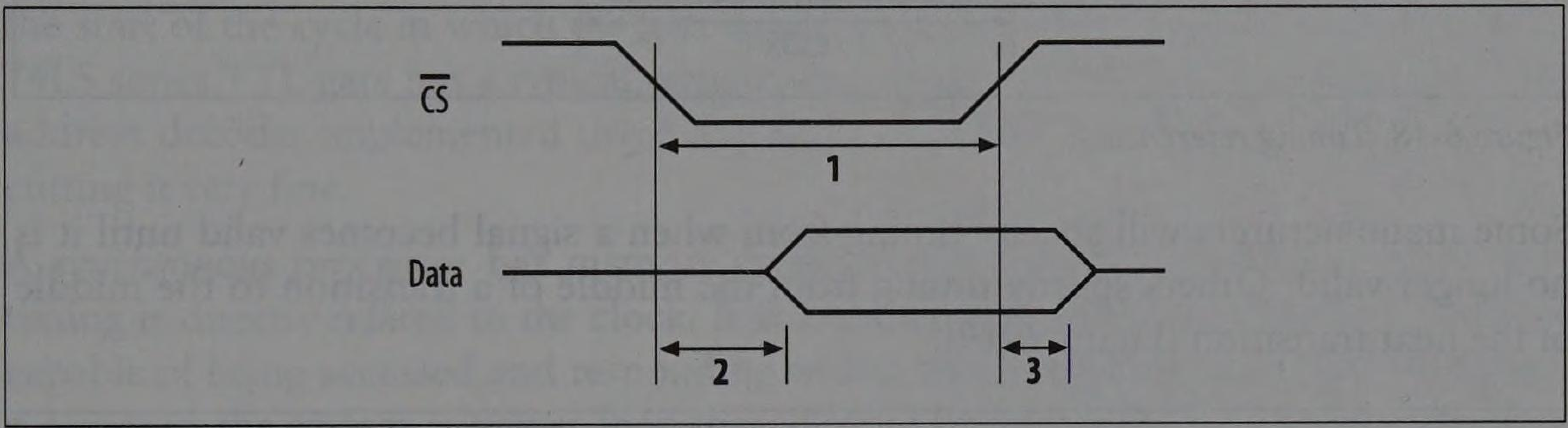


Figure 6-17. Example timing diagram

The diagram shows the relationship between input signals to the device (such as $\overline{\textbf{CS}}$) and outputs from the device (such as **Data**). The numbers on the diagram are references to timing information within tables. They do not represent timing directly. Table 6-2 shows how a datasheet might list the timing parameters.

Table 6-2. Example timing parameters

Ref	Description	Min	Max	Units
1	$\overline{CS}$ hold time	60		ns
2	$\overline{CS}$ to data valid		30	ns
3	Data hold time	5	10	ns

Timing reference 1 (the first row of Table 6-2) shows how long $\overline{\textbf{CS}}$ must be held low. In this instance, it is a minimum of 60ns. This means that the device won't guarantee that $\overline{\textbf{CS}}$ will be recognized unless it is held low for more than this time. There is no maximum specified. This means that it doesn't matter if $\overline{\textbf{CS}}$ is held low for longer than 60ns. The only requirement is that it is low for a minimum of 60ns.

Timing reference 2 shows how long it takes the device to respond to $\overline{\textbf{CS}}$ going low. From when $\overline{\textbf{CS}}$ goes low until this device starts outputting data is a maximum of 30ns. What this means is that 30ns after $\overline{\textbf{CS}}$ goes low, the device will be driving valid

data onto the data bus. It may start driving data earlier than 30ns. The only guarantee is that it will take no longer than 30ns for this device to respond.

Timing reference 3 specifies when the device will stop driving data once $\overline{CS}$ has been negated. This reference has a minimum of 5ns and a maximum of 10ns. This means that data will be held valid for at least 5ns, but no more than 10ns, after $\overline{CS}$ negates.

Some manufacturers use numbers to reference timing, others may use labels (Figure 6-18).

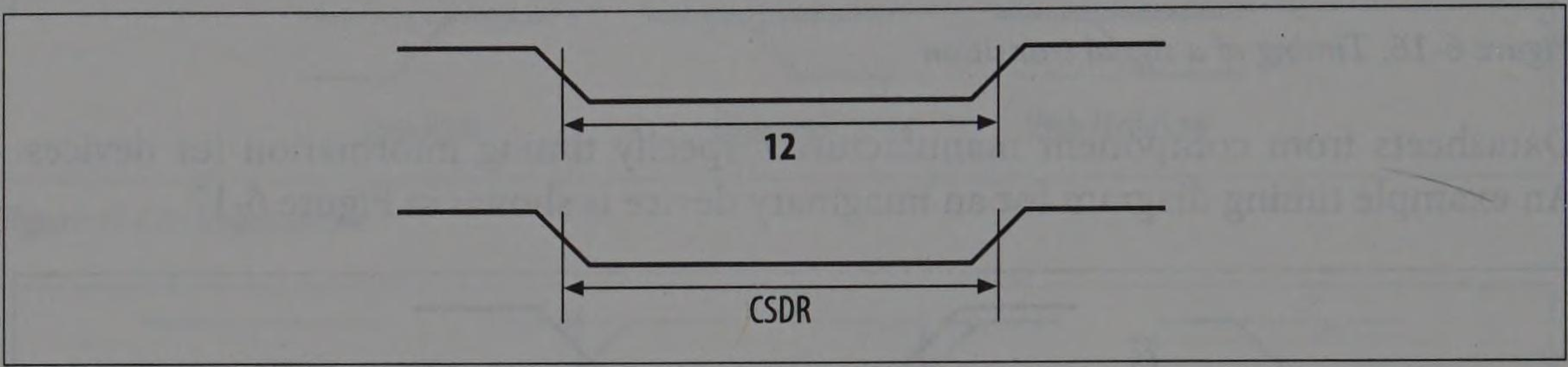


Figure 6-18. Timing reference

Some manufacturers will specify timing from when a signal becomes valid until it is no longer valid. Others specify timing from the middle of a transition to the middle of the next transition (Figure 6-19).

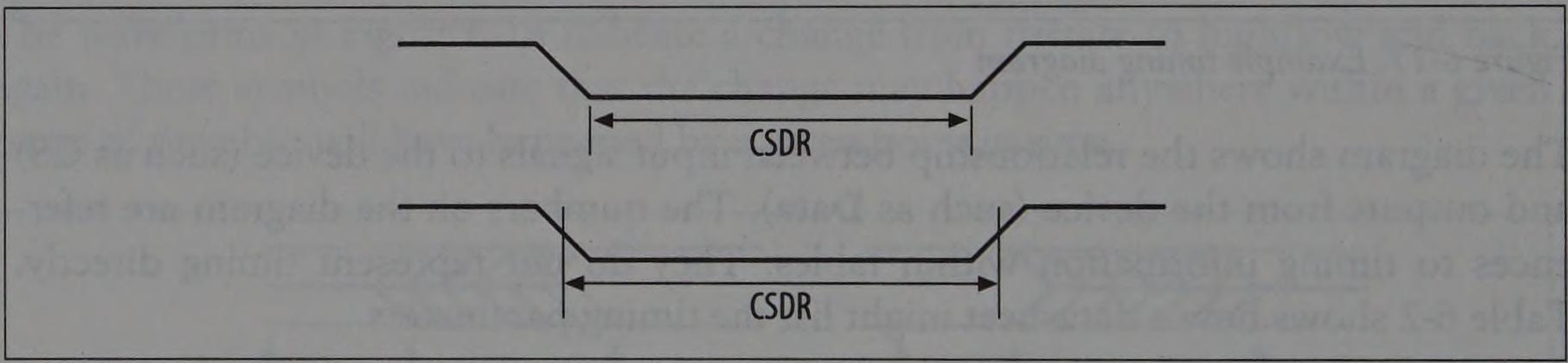


Figure 6-19. Timing length

So, with all that in mind, let's look at the timing for a real processor. Different processor architectures have different signals and different timing, but once you understand one, the basic principles can be applied to all. Since most small microcontrollers don't have external buses, the choice is very limited. We'll look at the one, and only, AVR with an external bus—the AT90S8515. In the PIC world, the PIC17C44 is capable of bus-based interfacing.

AT90S8515 Memory Cycle

A *memory cycle* (also known as a *machine cycle* or *processor cycle*) is defined as the period of time it takes for a processor to initiate an access to memory (or peripheral), perform the transfer, and terminate the access. The memory cycle generated by a processor is usually of a fixed period of time (or multiples of a given period) and may take several (processor) clock cycles to complete.

Memory cycles usually fall into two categories, the *read cycle* and the *write cycle*. The memory or device that is being accessed requires that the data is held valid for a given period after it has been selected and after a read or write cycle has been identified. This places constraints on the system designer. There is a limited time in which any glue logic (interface logic between the processor and other devices) must perform its function, such as selecting which external device is being accessed. The setup times must be met. If they are not, the computer will not function. The glue logic that monitors the address from the processor and uniquely selects a device is known as an *address decoder*. We'll take a closer look at address decoders shortly.

Timing is probably the most critical aspect of computer design. For example, if a given processor has a 150ns cycle time and a memory device requires 120ns from when it is selected until when it has completed the transfer, this leaves only 30ns at the start of the cycle in which the glue logic can manipulate the processor signals. A 74LS series TTL gate has a typical propagation delay of 10ns. So, in this example, an address decoder implemented using any more than two 74LS gates (in sequence) is cutting it very fine.

A *synchronous* processor has memory cycles of a fixed duration, and all processor timing is directly related to the clock. It is assumed that all devices in the system are capable of being accessed and responding within the set time of the memory cycle. If a device in the system is *slower* than that allowed by the memory cycle time, logic is required to pause the processor's access, thus giving the slow device time to respond. Each clock cycle within this pause is known as a *wait state*. Once sufficient time has elapsed (and the device is ready), the processor is released by the logic and continues with the memory cycle. Pausing the processor for slower devices is known as *inserting wait states*. The circuitry that causes a processor to hold is known as a *wait-state generator*. A wait state generator is easily achieved using a series of flip-flops acting as a simple counter. The generator is enabled by a processor output indicating that a memory cycle is beginning and is normally reset at the end of the memory cycle to return it to a known state. (Some processors come with internal, programmable wait-state generators.)

An *asynchronous* processor does not terminate its memory cycle within a given number of clock cycles. Instead, it waits for a transfer acknowledge assertion from the device or support logic to indicate that the device being accessed has had sufficient time to complete its part in the memory cycle. In other words, the processor automatically inserts wait states in its memory cycle until the device being accessed is ready. If the processor does not receive an acknowledge, it will wait indefinitely. Many computer systems using asynchronous processors have additional logic to cause the processor to restart if it waits too long for a memory cycle to terminate. An asynchronous processor can be made into a synchronous processor by tying the acknowledge line to its active state. It then assumes that all devices are capable of keeping up with it. This is known as *running with no wait states*.

Most microcontrollers are synchronous, whereas most larger processors are asynchronous. The AT90S8515 is a synchronous processor, and it has an internal wait-state generator capable of inserting a single wait state.

Bus Signals

Figure 6-20 shows an AT90S8515 processor with support components. The AT90S8515 has an address bus, a data bus, and a control bus that it brings to the outside world for interfacing. Since this processor has a limited number of pins, these buses share pins with the digital I/O ports (port A and port B) of the processor. A bit in a control register determines whether these pins are I/O or bus pins. Now, a 16-bit address bus and an 8-bit data bus add up to 24 bits, but ports A and B have only 16 bits between them. So how does the processor fit 24 bits into 16? It multiplexes the lower half of the address bus with the data bus. At the start of a memory access, port A outputs address bits A0..A7. The processor provides a control line, **ALE** (Address Latch Enable), which is used to control a latch, such as a 74HCT573 (shown on the right in Figure 6-20). As **ALE** falls, the latch grabs and holds the lower address bits. At the same time, port B outputs the upper address bits, A8..A15. These are valid for the entire duration of the memory access. Once the latch has acquired the lower address bits, port A then becomes the data bus for information transfer between the processor and an external device. Also shown in Figure 6-20 are the crystal circuit, the In-System Programming port, decoupling capacitors for the processor's power supply, and net labels for other important signals.

The timing diagrams for an AT90S8515 are shown in Figure 6-21. The cycle T3 exists only when the processor's wait state generator is enabled.

Now, let's look at these signals in more detail. (We'll see later how you actually work with this information. For the moment, we're just going to "take a tour" of the timing diagrams.) The numbers for the timing information can be found in the datasheet, available from ATMEL's web site. Figure 6-22 shows the timing information as presented in the ATMEL datasheet, complete with timing references.

The references are looked up in the appropriate table in the processor's datasheet (Table 6-3).

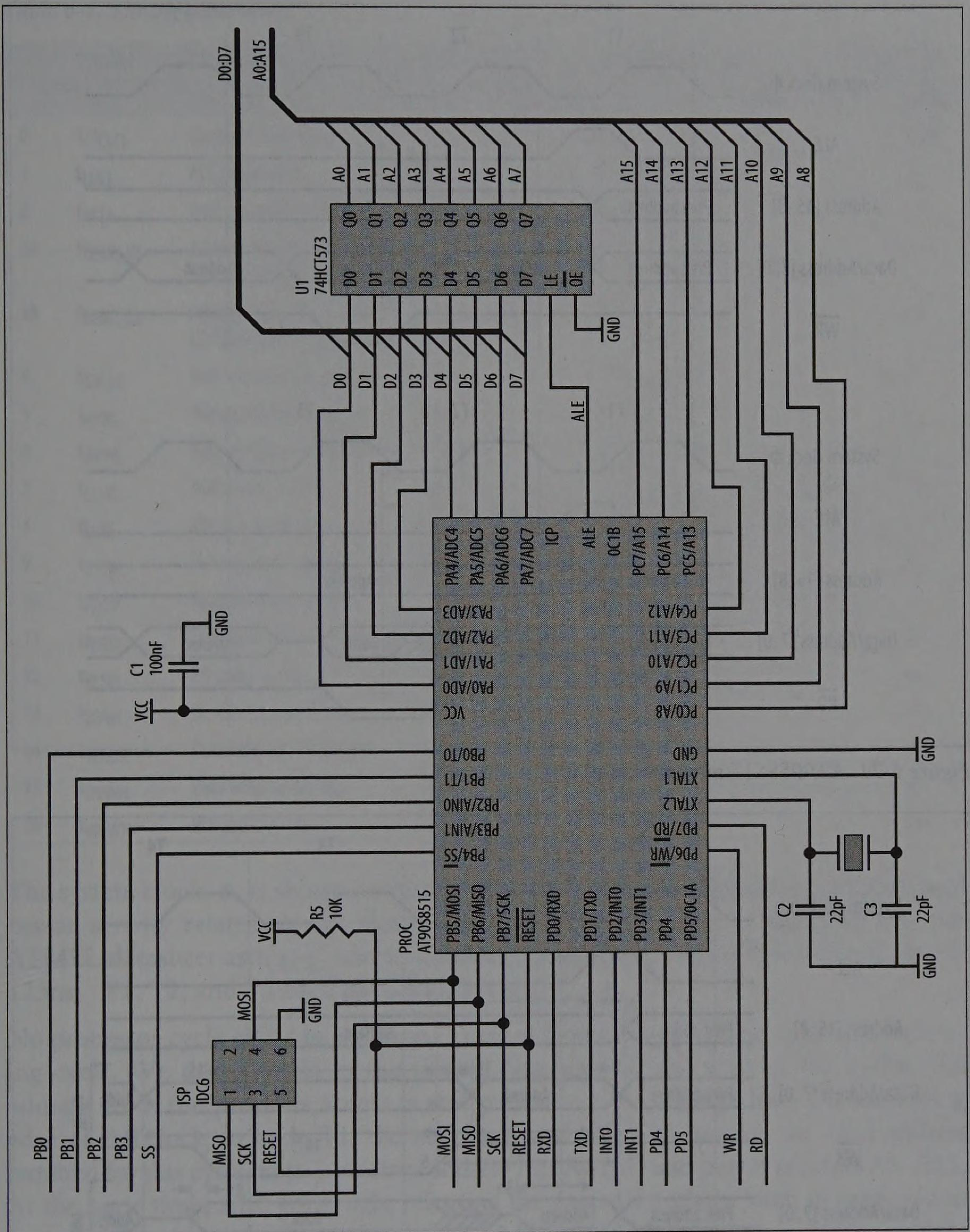


Figure 6-20. Address bus demultiplexing

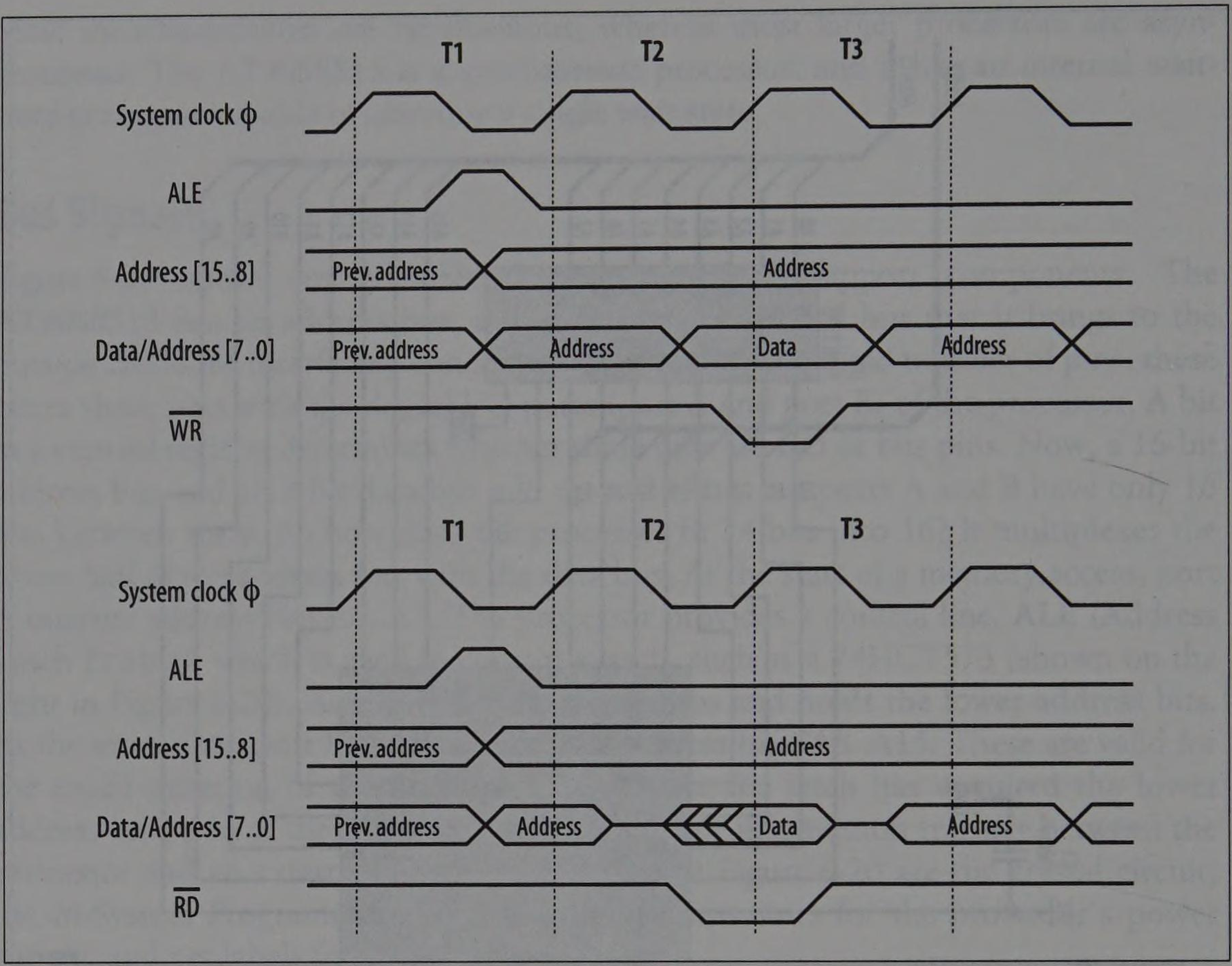


Figure 6-21. AT90S8515 memory cycles

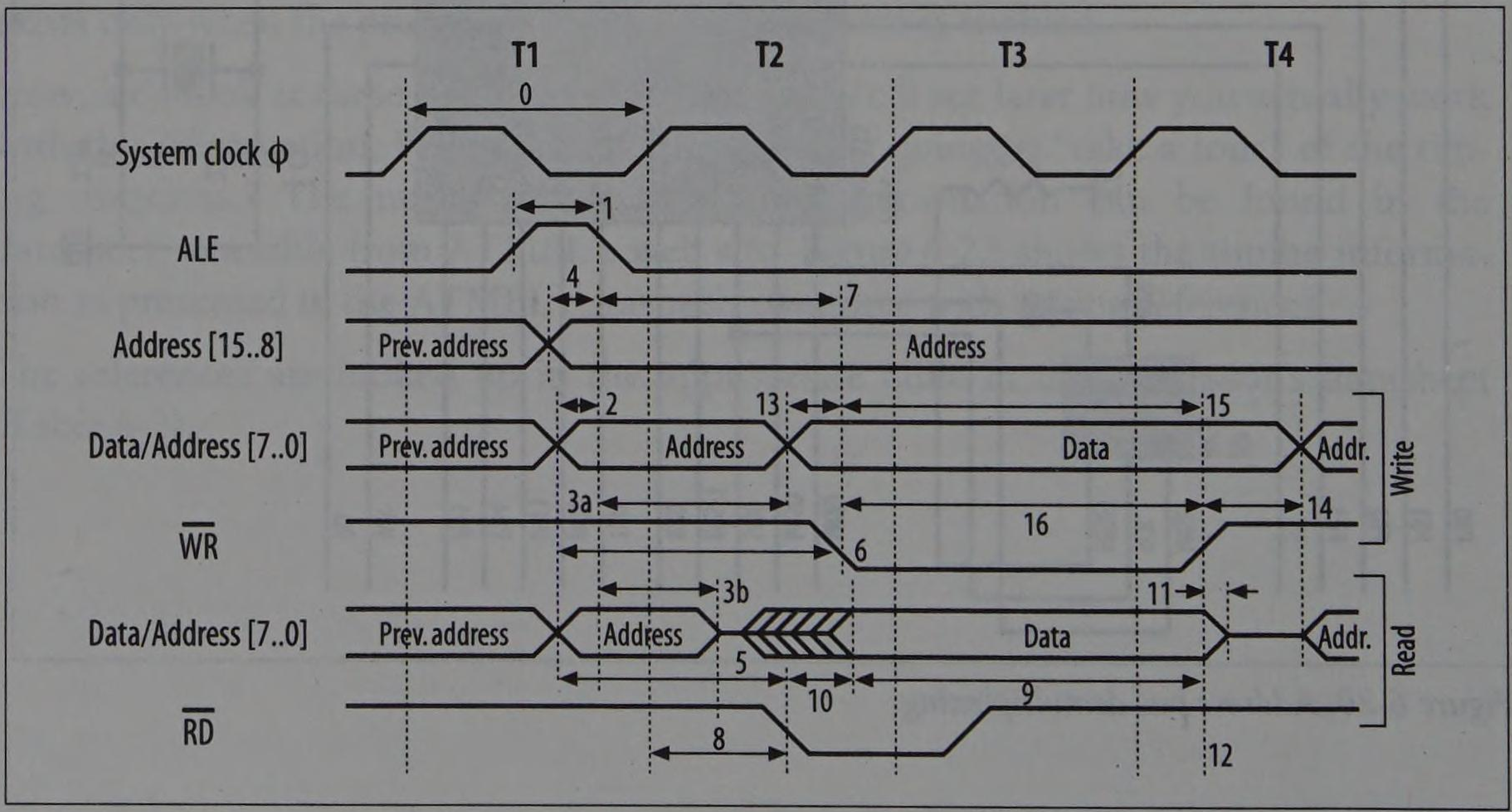


Figure 6-22. AT90S8515 memory cycles with timing parameters

Table 6-3. Timing parameters

	Symbol	Parameter	8MHz oscillator		Variable oscillator		
			Min	Max	Min	Max	Unit
0	$1/t_{CLCL}$	Oscillator frequency			0.0	8.0	MHz
1	t_{LHLL}	ALE pulse width	32.5		$0.5t_{CLCL}$-30.0		ns
2	t_{AVLL}	Address valid A to ALE low	22.5		$0.5t_{CLCL}$-40.0		ns
3a	t_{LLAX_ST}	Address hold after ALE low, ST/STD/STS instructions	67.5		$0.5t_{CLCL}$-50.0		ns
3b	t_{LLAX_LD}	Address hold after ALE low, LD/LDD/LDS instructions	15.0		15.0		ns
4	t_{AVLLC}	Address valid C to ALE low	22.5		$0.5t_{CLCL}$-40.0		ns
5	t_{AVRL}	Address valid to RD low	95.0		$1.0t_{CLCL}$-30.0		ns
6	t_{AVWL}	Address valid to WR low	157.5		$1.5t_{CLCL}$-30.0		ns
7	t_{LLWL}	ALE low to WR low	105.0	145.0	$1.0t_{CLCL}$-20.0	$1.0t_{CLCL}$+20.0	ns
8	t_{LLRL}	ALE low to RD low	42.5	82.5	$0.5t_{CLCL}$-20.0	$0.5t_{CLCL}$+20.0	ns
9	t_{DVRH}	Data setup to RD high	60.0		60.0		ns
10	t_{RLDV}	Read low to data valid		70.0		$1.0t_{CLCL}$-55.0	ns
11	t_{RHDX}	Data hold after RD high	0.0		0.0		ns
12	t_{RLRH}	PD pulse width	105.0		$1.0t_{CLCL}$-20.0		ns
13	t_{DVWL}	Data setup to WR low	27.5		$0.5t_{CLCL}$-35.0		ns
14	t_{WHDX}	Data hold after WR high	0.0		0.0		ns
15	t_{DVWH}	Data valid to WR high	95.0		$1.0t_{CLCL}$-30.0		ns
16	t_{WLWH}	WR pulse width	42.5		$0.5t_{CLCL}$-20.0		ns

The system clock, φ, is shown at the top of both diagrams for reference, since all processor activity relates to this clock. The period of the clock is designated in the ATMEL datasheet as t_{CLCL}* and is equal to 1/frequency. For an 8MHz clock, this is 125ns. T1, T2, and T3 each has a width of t_{CLCL}.

No processor cycle exists in isolation. There is always† a preceding cycle and following cycle. We can see this in the timing diagrams. At the start of the cycles, the address from the previous access is still present on the address bus. On the falling edge of the clock, in cycle T1, the address bus changes to become the valid address required for this cycle. Port A presents address bits A0..A7, and port B presents A8..A15. At the same time, **ALE** goes high, releasing the external address latch in preparation for acquiring the new address from port A. **ALE** stays high for `0.5 x tCLCL - 30ns`.

* Datasheet nomenclature can often be very cryptic. The *CL* comes from *clock*. Since Atmel uses four character subscripts for timing references, they pad by putting *CL* twice. You don't really need to know what the subscripts actually mean, you just need to know the signals they refer to and the actual numbers involved.

† I'm ignoring coming out of reset or just before power-off!

So, for example, with an AT90S8515 running at 8MHz, **ALE** stays high for 32.5ns. **ALE** falls, causing the external latch to acquire and hold the lower address bits. Prior to **ALE** falling, the address bits will have been valid for `0.5 x tCLCL - 40ns` or, in other words, 40ns before the system clock rises at the end of the T1 period. After **ALE** falls, the lower address bits will be held on port A for `0.5 x tCLCL + 5ns`, for a write cycle, before changing to data bits. For a read cycle, they are held for a minimum of 15ns only. The reason this is so much shorter for a read cycle is that the processor wishes to free those signal pins as soon as possible. Since this is a read cycle, an external device is about to respond, which means the processor needs to get out of the way as soon as it can.

For a write cycle, `tCLCL - 20ns` after **ALE** goes low, the write strobe, $\overline{\textbf{WR}}$, goes low. This indicates to external devices that the processor has output valid data on the data bus. $\overline{\textbf{WR}}$ will be low for `0.5 x tCLCL - 20ns`. This time is to allow the external device to prepare to read in (latch) the data. On the rising edge of $\overline{\textbf{WR}}$, the external device is expected to latch the data presented on the data bus. At this point, the cycle completes, and the next cycle is about to begin.

For a read cycle, the read strobe, $\overline{\textbf{RD}}$, goes low `0.5 x tCLCL - 20ns` after **ALE** is low. $\overline{\textbf{RD}}$ will be low for `tCLCL - 20ns`. During this time, the external device is expected to drive valid data onto the data bus. It can present data any time after $\overline{\textbf{RD}}$ goes low, as long as data is present and stable at least 60ns before $\overline{\textbf{RD}}$ goes high again. At this point, the processor latches the data from the external device, and the read cycle terminates. Note that many processors may not have a separate read enable signal, so this must be generated by external logic, based on the premise that if the cycle is not a write cycle, it must be a read cycle.

So, that is how an AT90S8515 expects to access any external device attached to its buses, whether those devices are memory chips or peripherals. But how does it work in practice? Let's look at designing* a computer based on an AT90S8515, with some external devices. For this example, we will interface the processor to a static RAM and some simple latches that we could use to drive banks of LEDs.

Memory Maps and Address Decoding

To the processor, its address space is one big linear region. Although there may be numerous devices within that space, both internal to the processor and external, it makes no distinction between devices. The processor simply performs memory accesses with the address space. It is up to the system designer (that's you) to allocate regions of memory to each device and then to provide address decode logic. The address decoder takes the address provided by the processor during an external access and uniquely selects the appropriate device (Figure 6-23). For example, if we

* Since we've covered oscillators and in-circuit programming previously, I'll ignore those in this discussion. That doesn't mean that you should leave them out of your design!

have a RAM occupying a region of memory, any address from the processor corresponding to within that region should select the RAM and *not* select any other device. Similarly, any address outside that region should leave the RAM unselected.

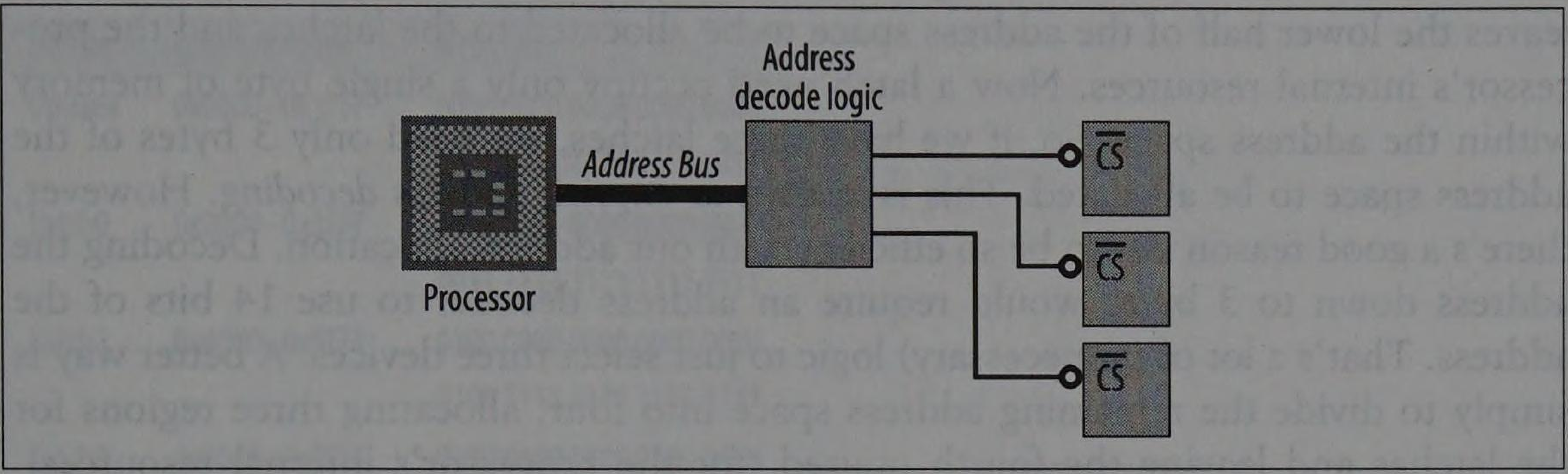


Figure 6-23. An address decoder uses the address to select one of several devices

The allocation of devices within an address space is known as a *memory map* or *address map*. The address spaces for an AT90S8515 processor are shown in Figure 6-24. Any device we interface to the processor must be within the data memory space. Thus, we can ignore the processor's internal program memory. As the processor is a Harvard architecture, the program space is a completely separate address space. Within the 64K data space lie the processor's internal resources—the working registers, the I/O registers and the internal 512 bytes of SRAM. These occupy the lowest addresses within the space. Any address above 0x0260 is ours to play with. (Not all processors have resources that are memory mapped, and in those cases the entire memory space is usable by external devices.)

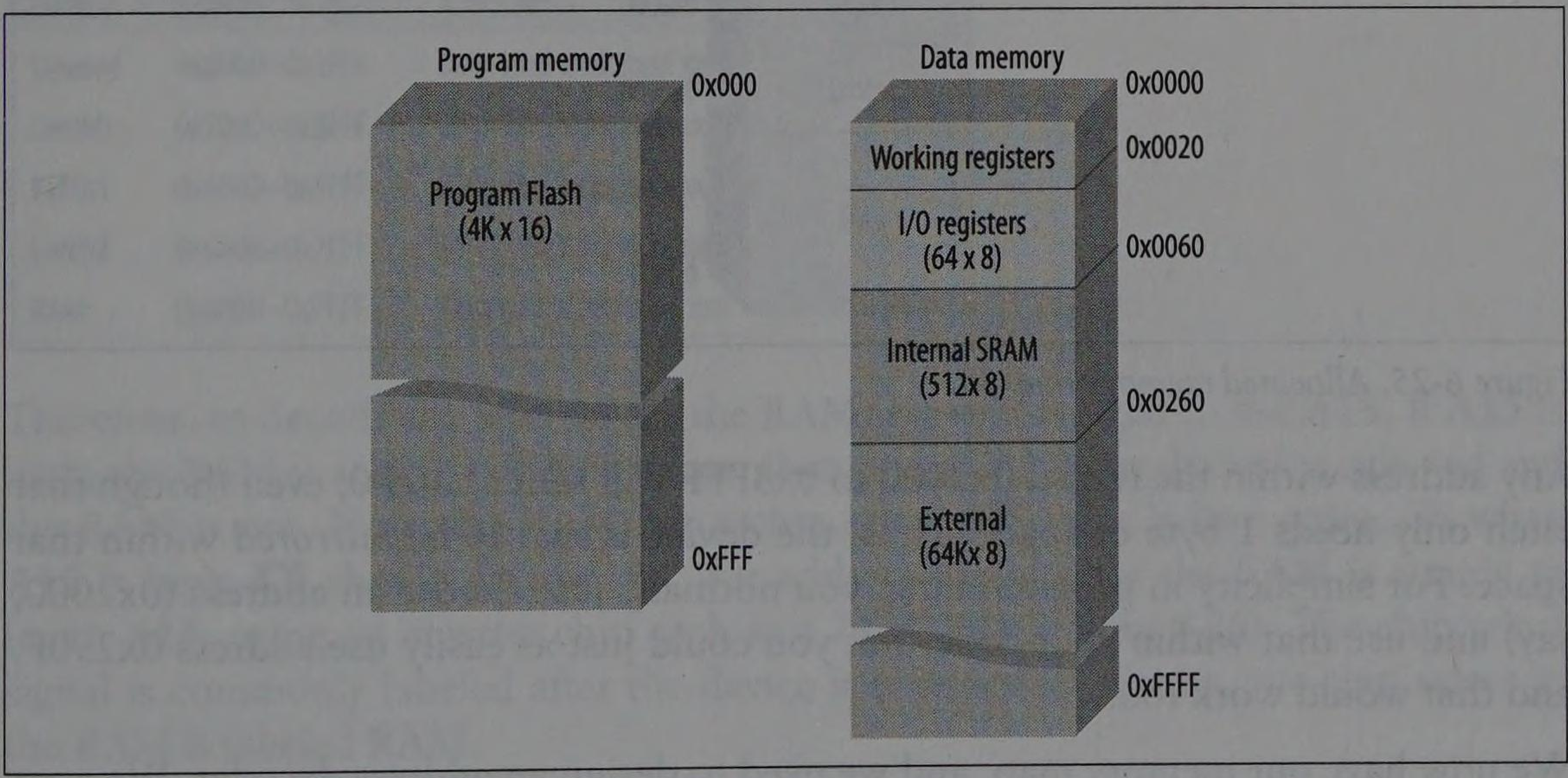


Figure 6-24. ATMEL AT90S8515 memory map

Now, our first task is to allocate the remaining space to the external devices. Since the RAM is 32K in size, it makes sense to place it within the upper half of the address space (0x8000–0xFFFF). Address decoding becomes much easier if devices are placed on neat boundaries. Placing the RAM between addresses 0x8000 and 0xFFFF leaves the lower half of the address space to be allocated to the latches and the processor's internal resources. Now a latch need occupy only a single byte of memory within the address space. So, if we have three latches, we need only 3 bytes of the address space to be allocated. This is known as *explicit address decoding*. However, there's a good reason not to be so efficient with our address allocation. Decoding the address down to 3 bytes would require an address decoder to use 14 bits of the address. That's a lot of (unnecessary) logic to just select three devices. A better way is simply to divide the remaining address space into four, allocating three regions for the latches and leaving the fourth unused (for the processor's internal resources). This is known as *partial address decoding* and is much more efficient. The trick is to use the minimal amount of address information to decode for your devices.

Our address map allocated to our static RAM and three latches is shown in Figure 6-25. Note that the lowest region leaves the addresses in the range 0x0260 to 0x1FFF unused.

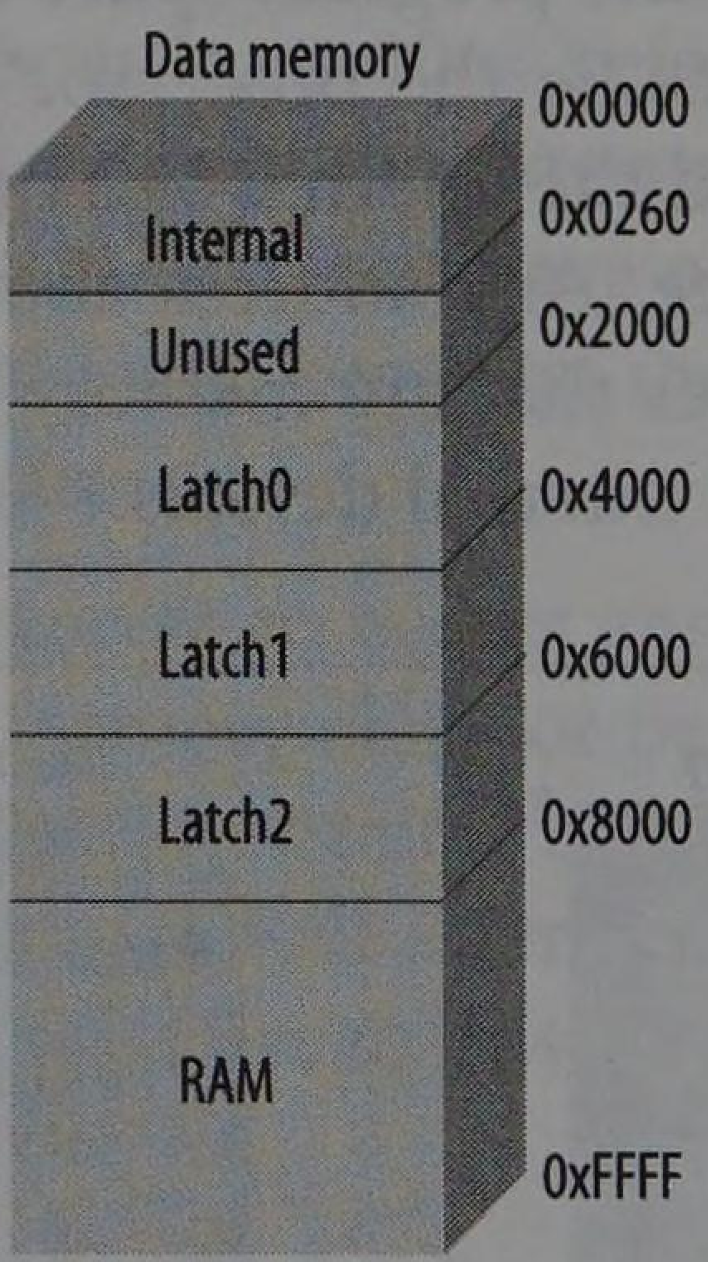


Figure 6-25. Allocated memory map

Any address within the region 0x2000 to 0x3FFF will select Latch0, even though that latch only needs 1 byte of space. Thus, the device is said to be *mirrored* within that space. For simplicity in programming, you normally just choose an address (0x2000, say) and use that within your code. But you could just as easily use address 0x290F, and that would work too.

We now have our memory map, and we need to design an address decoder. We start by tabling the devices, along with their addresses (Table 6-4). We need to look for

which address bits are different between the devices and which address bits are common within a given device's region.

Table 6-4. Address table

Device	Address range	A15 .. A0
Unused	0x0000–0x1FFF	0000 0000 0000 0000 0000
		0001 1111 1111 1111 1111
Latch0	0x2000–0x3FFF	0010 0000 0000 0000 0000
		0011 1111 1111 1111 1111
Latch1	0x4000–0x5FFF	0100 0000 0000 0000 0000
		0101 1111 1111 1111 1111
Latch2	0x6000–0x7FFF	0110 0000 0000 0000 0000
		0111 1111 1111 1111 1111
RAM	0x8000 – 0xFFFF	1000 0000 0000 0000 0000
		1111 1111 1111 1111 1111

So, what constitutes a unique address combination for each device? Looking at the table, we can see that for the RAM, address bit (and address signal) **A15** is high, while for every other device it is low. We can therefore use **A15** as the trigger to select the RAM. For the latches, address bits **A15**, **A14**, and **A13** are critical. So we can redraw our table to make it clearer. (This is the more common way of doing an address table—Table 6-5.) An x means a "don't-care" bit.

Table 6-5. Simplified address table

Device	Address range	A15 .. A0
Unused	0x0000–0x1FFF	000x xxxx xxxx xxxx xxxx
Latch0	0x2000–0x3FFF	001x xxxx xxxx xxxx xxxx
Latch1	0x4000–0x5FFF	010x xxxx xxxx xxxx xxxx
Latch2	0x6000–0x7FFF	011x xxxx xxxx xxxx xxxx
RAM	0x8000–0xFFFF	1xxx xxxx xxxx xxxx xxxx

Therefore, to decode the address for the RAM, we simply need to use **A15**. If **A15** is high, the RAM is selected. If **A15** is low, then one of the other devices is selected and the RAM is not. Now, the RAM has a chip select ($\overline{\textbf{CS}}$) that is low active. So when **A15** is high, $\overline{\textbf{CS}}$ should go low. So, our address decoder for the RAM is simply to invert **A15**, using an inverter chip such as a 74HCT04 (Figure 6-26). The chip select signal is commonly labeled after the device it is selecting. Hence, our chip select to the RAM is labeled $\overline{\textbf{RAM}}$.

Note that for the RAM to respond, it needs both a chip select and either a read or write strobe from the processor. All other address lines from the processor are connected directly to the corresponding address inputs of the RAM (Figure 6-27).

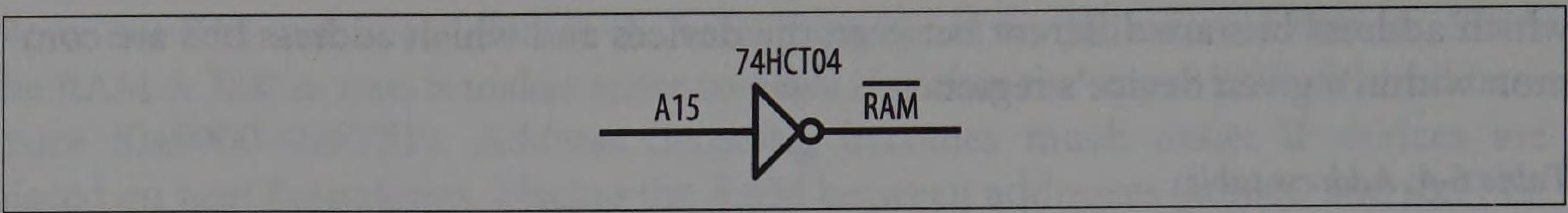


Figure 6-26. Address decoder for the RAM

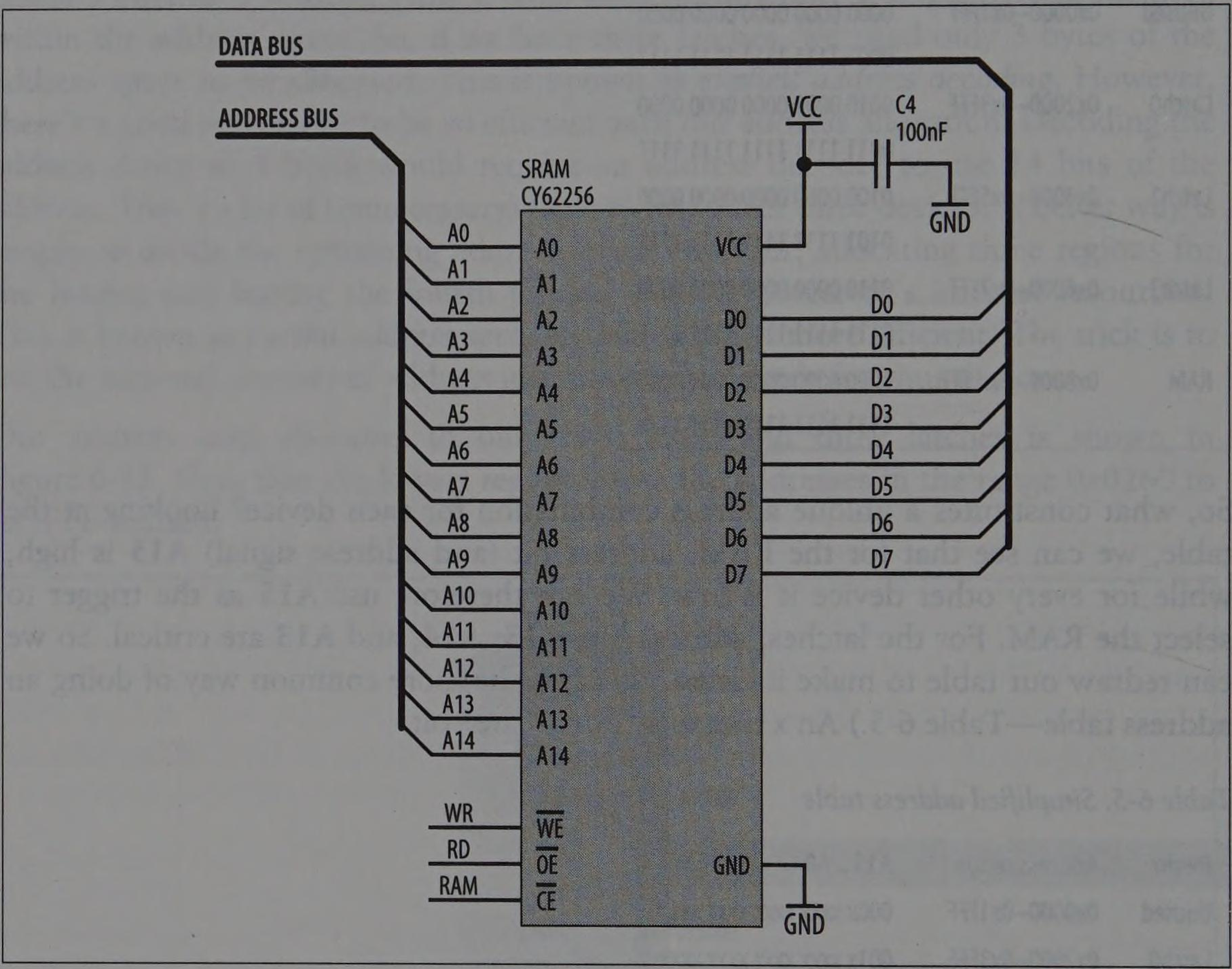


Figure 6-27. Connections to the SRAM

Now, for the other four regions, **A15** must be low, and **A14** and **A13** are sufficient to distinguish between the devices. Our address decoder, using discrete logic, would need several gates and would be messy. There's a simpler way. We can use a 74HCT139* decoder, which will take two address inputs (A and B) and gives us four unique, low-active chip select outputs (labeled Y0..Y3). So, our complete address decoder for the computer is shown in Figure 6-28.

The 74HCT139 uses **A15** (low) as an enable (input $\overline{G}$), and in this way, **A15** is included as part of the address decode. If we needed to decode for eight regions instead of four, we could have used a 74HCT138 decoder, which takes three address inputs and gives us eight chip selects.

* There are actually two separate decoders in each 74HCT139 chip. We'll only need one.

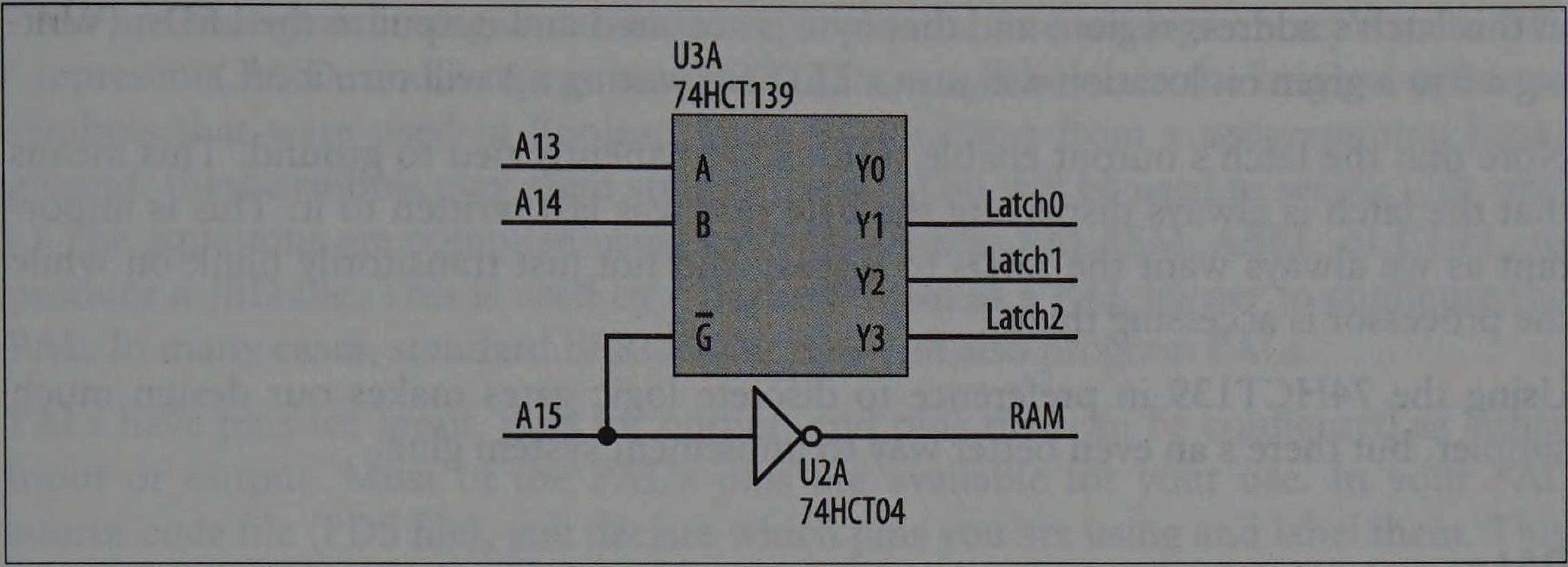


Figure 6-28. Complete address decoder

The interface between the processor and an output latch is simple. We can use the same type of latch (a 74HCT573) that we used to demultiplex the address. Such an output latch could be used in any situation in which we need some extra digital outputs. In the example circuit shown in Figure 6-29, I'm using the latch to control a bank of eight LEDs.

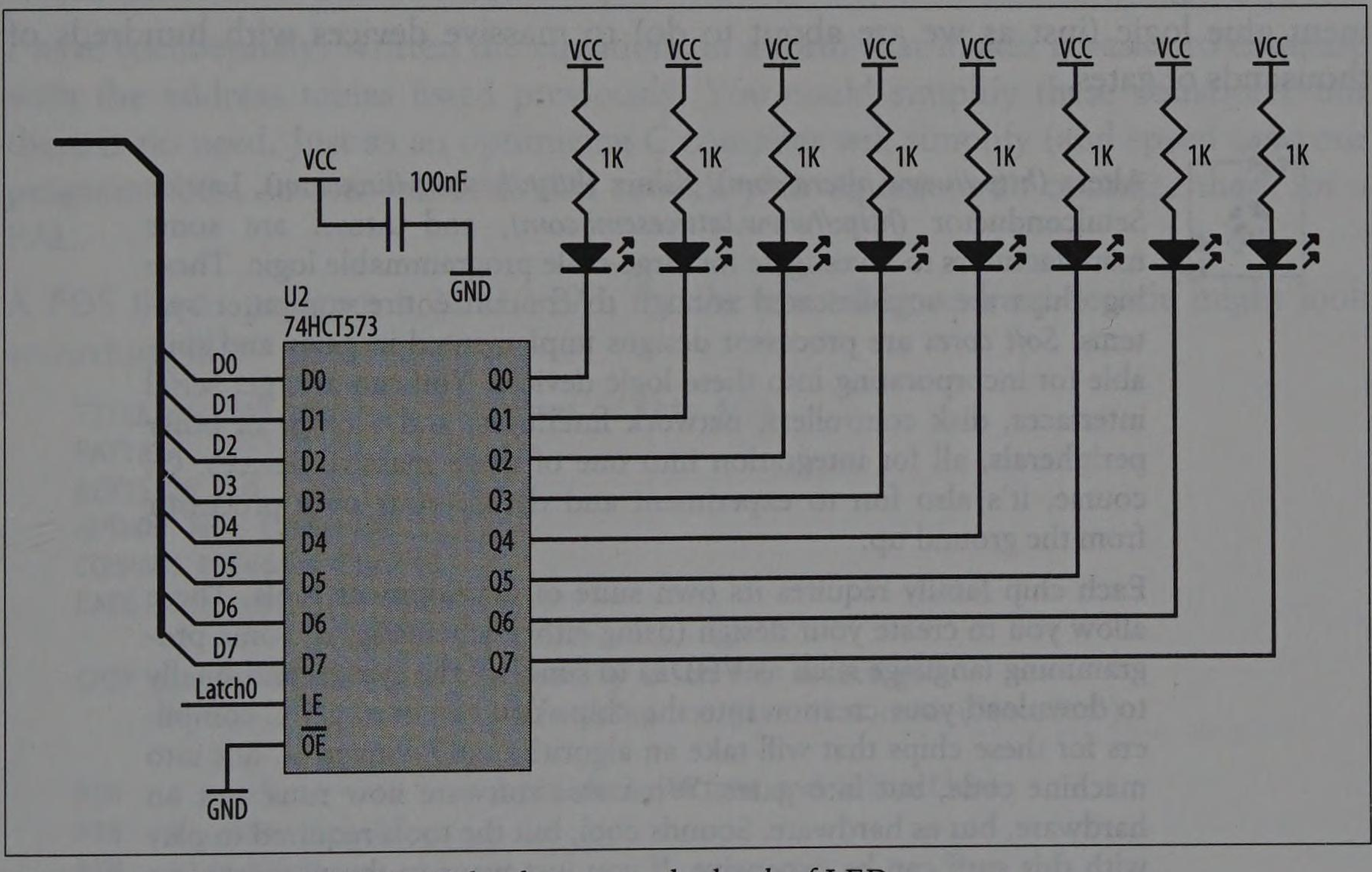


Figure 6-29. Using a 74HCT573 latch to control a bank of LEDs

The output from our 74HCT139 address decoder is used to drive the LE (Latch Enable) input of the 74HCT573. Whenever the processor accesses the region of memory space allocated to this device, the address decoder triggers the latch to acquire whatever is on the data bus. And, so, the processor simply writes a byte to any address

in this latch's address region, and that byte is acquired and output to the LEDs. (Writing a 0 to a given bit location will turn a LED on; writing a 1 will turn it off.)

Note that the latch's output enable ($\overline{\text{OE}}$) is permanently tied to ground. This means that the latch is always displaying the byte that was last written to it. This is important as we always want the LEDs to display and not just transitorily blink on while the processor is accessing them.

Using the 74HCT139 in preference to discrete logic gates makes our design much simpler, but there's an even better way to implement system glue.

PALs

Support logic is rarely implemented using individual gates. More common is programmable logic (PALs, LCAs, or PLDs)* to implement the miscellaneous glue functions that a computer system requires. Such devices are fast, take up relatively little space, have low power consumption, and as they are reprogrammable, make system design much easier and more versatile.

A wide range of devices is available, from simple chips that can be used to implement glue logic (just as we are about to do) to massive devices with hundreds of thousands of gates.

> Altera (*http://www.altera.com*), Xilinx (*http://www.xilinx.com*), Lattice Semiconductor (*http://www.latticesemi.com)*, and Atmel are some manufacturers to investigate for large-scale programmable logic. These big chips are sophisticated enough to contain entire computer systems. *Soft cores* are processor designs implemented in gates and suitable for incorporating into these logic devices. You can also get serial interfaces, disk controllers, network interfaces, and a range of other peripherals, all for integration into one of these massive devices. Of course, it's also fun to experiment and design your own processor from the ground up.
>
> Each chip family requires its own suite of development tools. These allow you to create your design (using either schematics or some programming language such as VHDL) to simulate the system and finally to download your creation into the chip. You can even get C compilers for these chips that will take an algorithm and convert it, not into machine code, but into gates. What was software now runs, not on hardware, but as hardware. Sounds cool, but the tools required to play with this stuff can be expensive. If you just want to throw together a small, embedded system, they are probably out of your price range. For what we need to do for our glue logic, such chips are overkill.

Since our required logic is simple, we will use a simple (and cheap) PAL that can be programmed using freely available, public-domain software.

* Programmable Array Logic, Logic Cell Arrays, and Programmable Logic Devices, respectively.

PALs are configured using equations to represent the internal logic: + represents OR, * represents AND, and / represents NOT. (These symbols are the original operator symbols that were used in Boolean logic. If you come from a programming background, these symbols may seem strange to you. You will be used to seeing |, &, and !.) The equations are compiled using software such as PALASM, ABEL, or CUPL, to produce a JED file. This is used by a device known as a *PAL burner* to configure the PAL. In many cases, standard EPROM burners will also program PALs.

PALs have pins for input, pins for output, and pins that can be configured as either input or output. Most of the PAL's pins are available for your use. In your PAL source code file (PDS file), you declare which pins you are using and label them. This is not unlike declaring variables in program source code, except that instead of allocating bytes of RAM, you're allocating physical pins of a chip. You then use those pin labels within equations to specify the internal logic. Our address decoder, implemented in a PAL, would have the following equations to specify the decode logic:

```
RAM = /A15
LATCH0 = /(/A15 * /A14 * A13)
LATCH1 = /(/A15 * A14 * /A13)
LATCH2 = /(/A15 * A14 * A13)
```

I have (deliberately) written the equations in a form that makes it easier to compare with the address tables listed previously. You could simplify these equations, but there is no need. Just as an optimizing C compiler will simplify (and speed up) your program code, so too will PALASM rework your equations to optimize them for a PAL.

A PDS file to program a 22V10 PAL for the preceding address decode might look something like:

```
TITLE decoder.pds          ; name of this file
PATTERN
REVISION 1.0
AUTHOR John Catsoulis
COMPANY Embedded Pty Ltd
DATE June 2002

CHIP decoder PAL22V10      ; specify which PAL device you
                           ; are using and give it a name ("decoder")

PIN   2   A15              ; pin declarations and allocations
PIN   3   A14
PIN   12  LATCH0
PIN   13  LATCH1
PIN   14  LATCH2
PIN   15  RAM

EQUATIONS                  ; equations start here

RAM = /A15
LATCH0 = /(/A15 * /A14 * A13)
```

```
LATCH1 = /(/A15 * A14 * /A13)
LATCH2 = /(/A15 * A14 * A13)
```

The advantages of using a PAL for system logic are two-fold. The PAL equations may be changed to correct for bugs or design changes. The propagation delays through the PAL are of a fixed and small duration (no matter what the equations), which makes analyzing the overall system's timing far simpler. For very simple designs, it probably doesn't make a lot of difference whether you use PALs or individual chips. However, for more complicated designs, programmable logic is the only option. If you can use programmable logic devices in preference to discrete logic chips, please do so. They make life much easier.

Timing Analysis

Now that we have finished our logic design, the question is: will it actually work? It's time (pardon the pun) to work through the numbers and analyze the timing. This is the least fun, and most important, part of designing a computer.

We start with the signals (and timing) of the processor, then add in the effects of our glue logic, and finally see if this falls within the requirements of the device to which we are interfacing. We'll work through the example for the SRAM. For the other devices, the analysis follows the same method. The timing diagram for a read cycle for the SRAM is shown in Figure 6-30. The RAM I have chosen is a CY62256-70 (32K) SRAM made by Cypress Semiconductor. Most 32K SRAMs follow the JEDEC standard, which means that their pinouts and signals are all compatible. So, what works for one 32K SRAM should work for them all. But, the emphasis is on *should*, and, as always, check the datasheet for the individual device you are using.

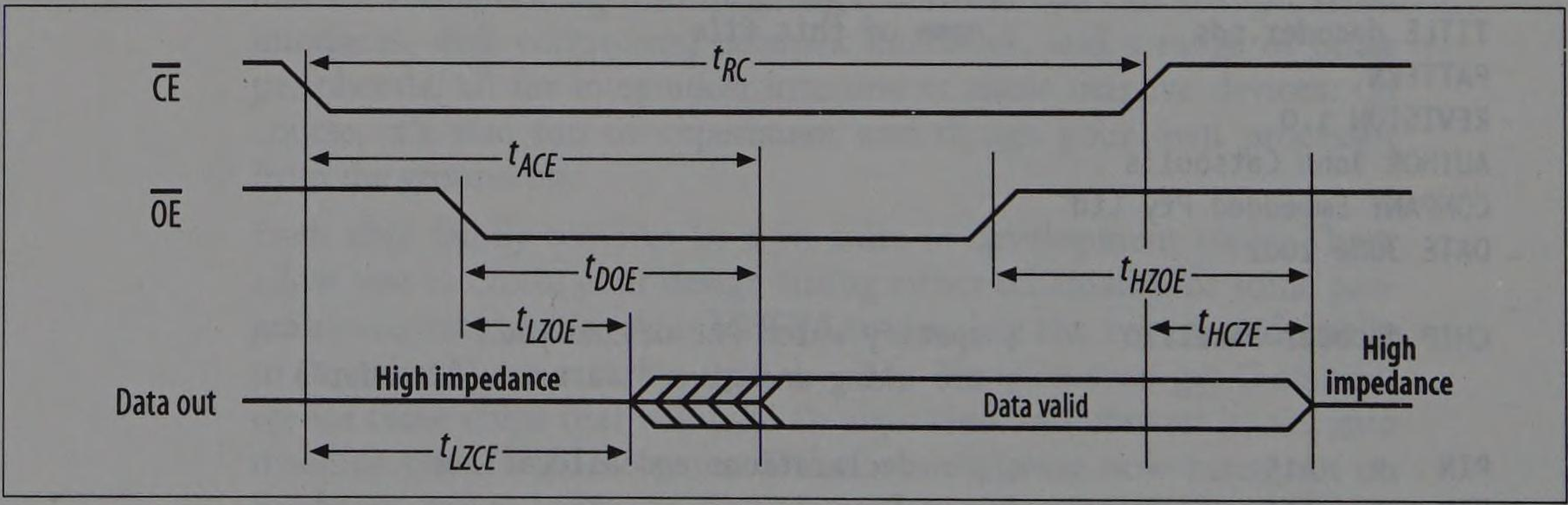


Figure 6-30. Timing for a read cycle to the RAM

The -70 in the part number means that this is a 70ns SRAM or, put simply, the access time for the chip is 70ns. Now, from the CY62256-70 datasheet (available from *http://www.cypress.com*), t_{RC} is a minimum of 70ns. This means that the chip enable, $\overline{\text{CE}}$, can be low for no less than 70ns. $\overline{\text{CE}}$ is just our chip select ($\overline{\text{RAM}}$) from our address decoder, and so we need to ensure that the address decoder will hold $\overline{\text{RAM}}$ low for at least this amount of time. For the SRAM to output data during a

read cycle, it needs a valid address, an active chip enable, and an active output enable ($\overline{\mathbf{OE}}$). The output enable is just the read strobe ($\overline{\mathbf{RD}}$) from the processor. These three conditions must be met before the chip will respond with data. It will take 70ns from $\overline{\mathbf{CE}}$ low (t_{ACE}) *or* 35ns from $\overline{\mathbf{OE}}$ low (t_{DOE}), whichever is the latter, until data is output. Now, $\overline{\mathbf{CE}}$ is generated by our address decoder (which in turn uses address information from the processor), and $\overline{\mathbf{OE}}$ ($\overline{\mathbf{RD}}$) comes from the processor. During a read cycle, the processor will output a read strobe and an address, which in turn will trigger the address decoder. Some time later in the cycle, the processor will expect data from the RAM to be present on the data bus. It is critical that the signals that cause the RAM to output data will do so in such a way that there will be valid data when the processor expects it. Meet this requirement and you have a processor that can read from external memory. Fail this requirement, and you'll have an intriguing paperweight and a talking piece at parties.

We start with the processor. I'm assuming that the processor's wait-state generator is disabled. For an AT90S8515 processor, everything is referenced to the falling edge of **ALE**. The high-order address bits, which feed our address decoder, become valid 22.5ns prior to **ALE** going low on an 8MHz AT90S8515. If we're using an address decoder, that takes 40ns* to respond to a change in inputs, our chip select for the RAM will become valid 17.5ns after **ALE** has fallen (Figure 6-31).

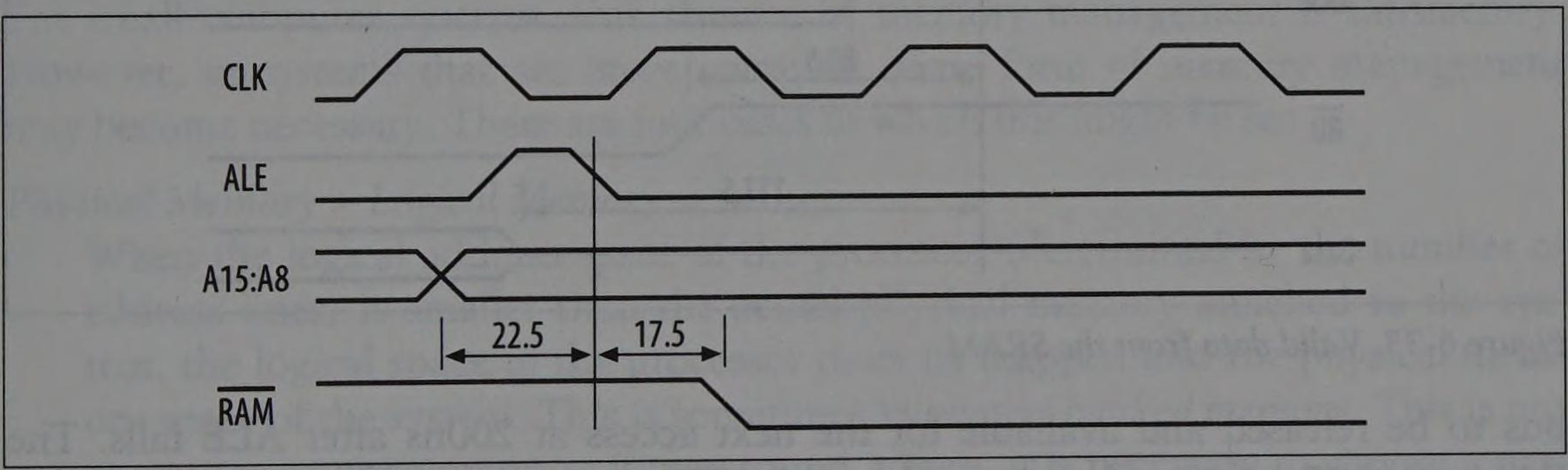


Figure 6-31. Timing for RAM chip select

Now, $\overline{\mathbf{RD}}$ will go low between 42.5ns and 82.5ns after **ALE** falls. Since the RAM will not output data until $\overline{\mathbf{RD}}$ ($\overline{\mathbf{OE}}$) is low, we take the worst case of 82.5ns (Figure 6-32).

The RAM will respond 70ns after $\overline{\mathbf{RAM}}$ and 35ns after $\overline{\mathbf{RD}}$, whichever is the last. So, 70ns from $\overline{\mathbf{RAM}}$ low is 87.5ns after **ALE**, and 35ns after $\overline{\mathbf{RD}}$ is 117.5ns after **ALE**. Therefore, $\overline{\mathbf{RD}}$ is the determining control signal in this case. This means that the SRAM will output valid data 117.5ns after **ALE** falls (Figure 6-33).

Now, an 8MHz processor expects to latch valid data during a read cycle at 147.5ns after **ALE**. So our SRAM will have valid data ready with 30ns to spare. So far, so good. But what about at the end of the cycle? Now, the processor expects the data

* PALs may respond in 15ns or less. This is another reason why PALs are a better choice than discrete logic.

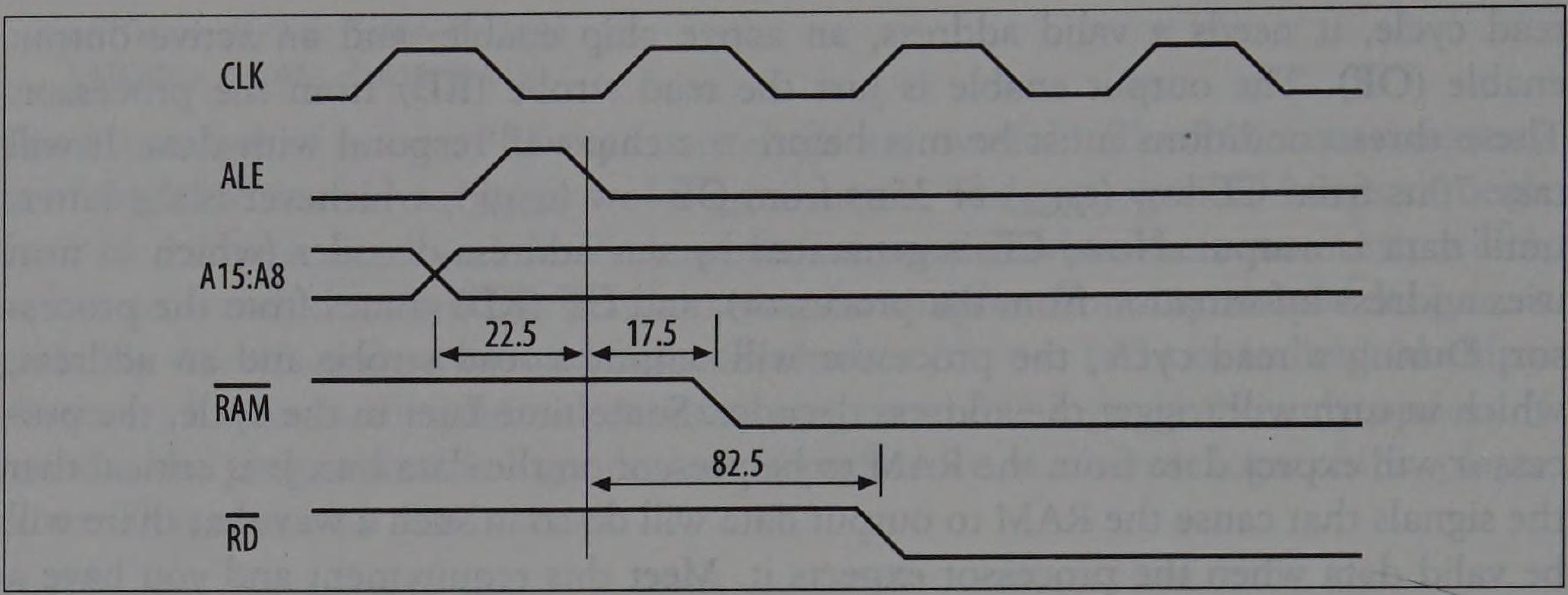


Figure 6-32. Read strobe and chip select for RAM

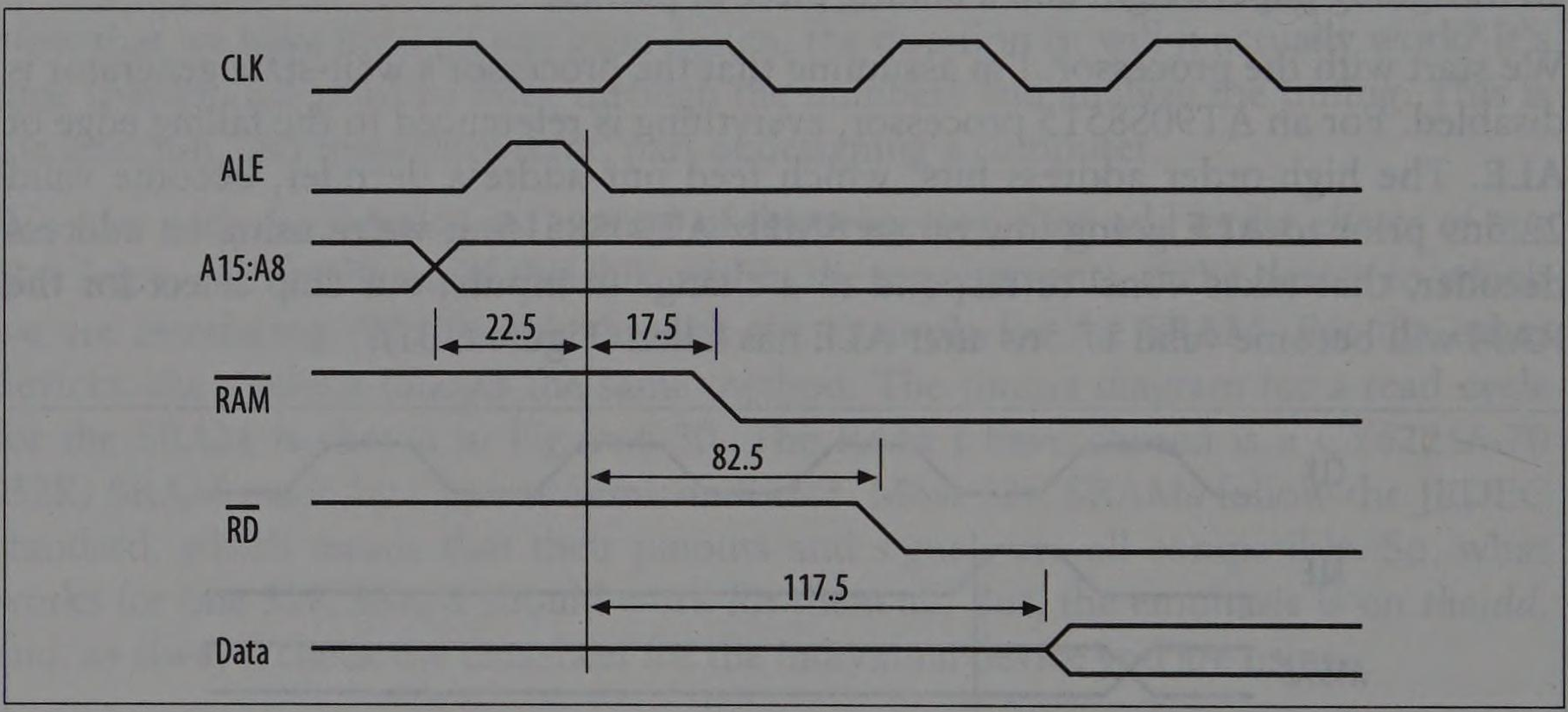


Figure 6-33. Valid data from the SRAM

bus to be released and available for the next access at 200ns after **ALE** falls. The RAM takes 25ns from when it is released by $\overline{\textbf{RD}}$ until it stops driving data onto the bus. This means that the data bus will be released by the RAM at 142.5ns. So that will work too.

The analysis for a write cycle is done in a similar manner. It is important to do this type of analysis for every device interfaced to your processor, for every type of memory cycle. It can be difficult, for datasheets are notorious for leaving information out, or presenting necessary data in a roundabout way. Working through it all can be time-consuming and frustrating, and it's far too easy to make a mistake. However, it is very necessary. Without it, you're relying on blind luck to make your computers go, and that's not good engineering.

Memory Management

In most small-scale embedded applications, the connections between a processor and an external memory chip are straightforward. Sometimes, though, playing with the natural order of things is advantageous. This is the realm of memory management.

Memory management deals with the translation of logical addresses to physical addresses and vice versa. A *logical address* is the address output by the processor. A *physical address* is the actual address being accessed in memory. In small computer systems, these are often the same. In other words, no address translation takes place, as illustrated in Figure 6-34.

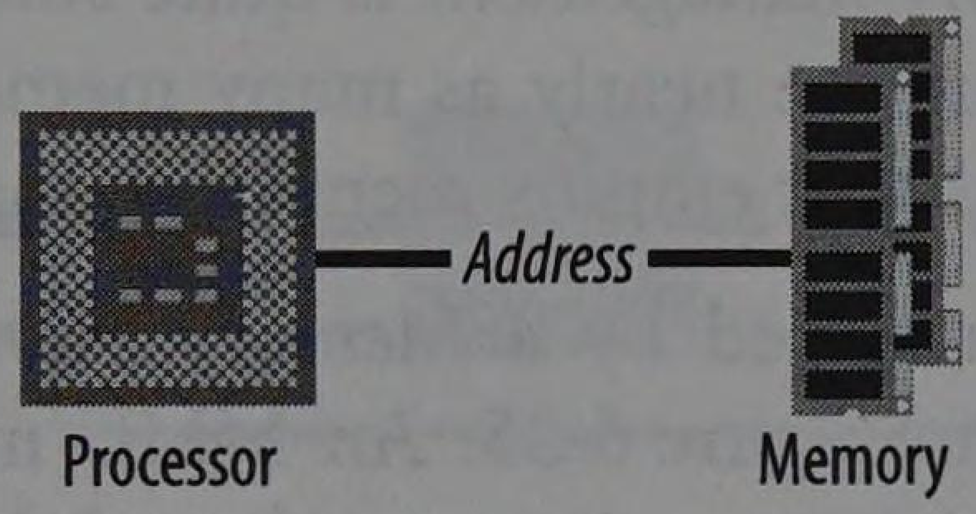


Figure 6-34. No address translation

For small computer systems, this absence of memory management is satisfactory. However, in systems that are more complex, some form of memory management may become necessary. There are four cases in which this might be so:

Physical Memory > Logical Memory
: When the logical address space of the processor (determined by the number of address lines) is smaller than the actual physical memory attached to the system, the logical space of the processor must be mapped into the physical memory space of the system. This is sometimes known as *banked memory*. This is not as strange or uncommon as it may sound. Often, it is necessary to choose a particular processor for a given attribute, yet that processor may have a limited address space: too small for the application. By implementing banked memory, the address space of the processor is expanded beyond the limitation of the logical address range.

Logical Memory > Physical Memory
: When the logical address space of the processor is very large, filling this address space with physical memory is not always practical. Some space on disk may be used as virtual memory, thus making the processor appear to have more physical memory than exists within the chips. Memory management is used to identify whether a memory access is to physical memory or virtual memory and must be capable of swapping the virtual memory on disk with real memory and performing the appropriate address translation.

Memory Protection

You may want to prevent some programs from accessing certain sections of memory. Protection can prevent a crashing program from corrupting the operating system and bringing down the computer. It is also a way of channeling all I/O access via the operating system, since protection can be used to prevent all software (save the OS) from accessing the I/O space.

Task Isolation

In a multitasking system, tasks should not be able to corrupt each other (by stomping on each other's memory space, for example). In addition, two separate tasks should be able to use the same logical address in memory with memory management performing the translation to separate, physical addresses.

The basic idea behind memory management is quite simple, but the implementation can be complicated, and there are nearly as many memory management techniques as there are computer systems that employ memory management.

Memory management is performed by a *Memory Management Unit (MMU)*. The basic form of this is shown in Figure 6-35. An MMU may be a commercial chip, a custom-designed chip (or logic), or an integrated module within the processor. Most modern, fast processors incorporate MMUs on the same chip as the CPU.

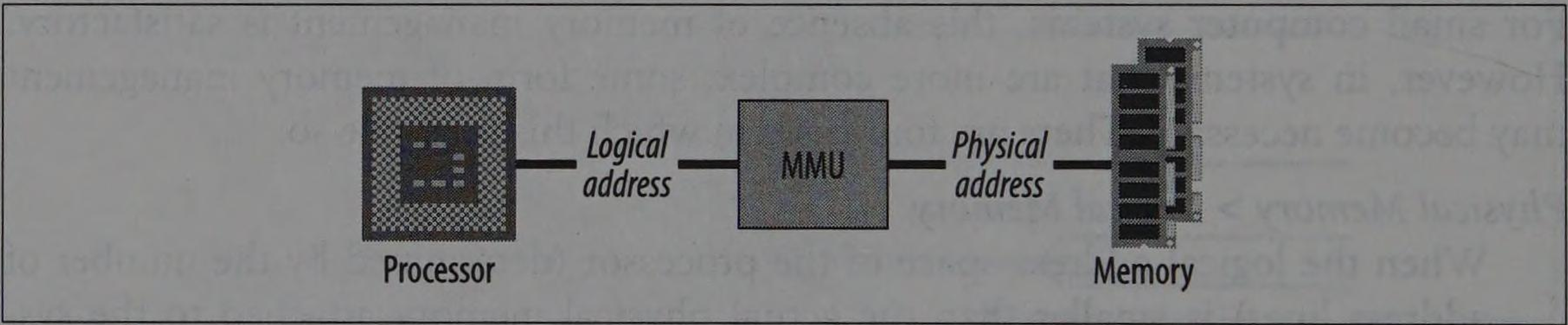


Figure 6-35. Address translation using an MMU

Page mapping

In all practical memory management systems, words of memory are grouped together to form pages, and an address can be considered to consist of a page number and the number of a word within that page. The MMU translates the logical page to a physical page while the word number is left unchanged (Figure 6-36). In practice, the overall address is just a concatenation of the page number and the word number.

The logical address from the processor is divided into a page number and a word number. The page number is translated by the MMU and recombined with the word number to form the physical address presented to memory (Figure 6-37).

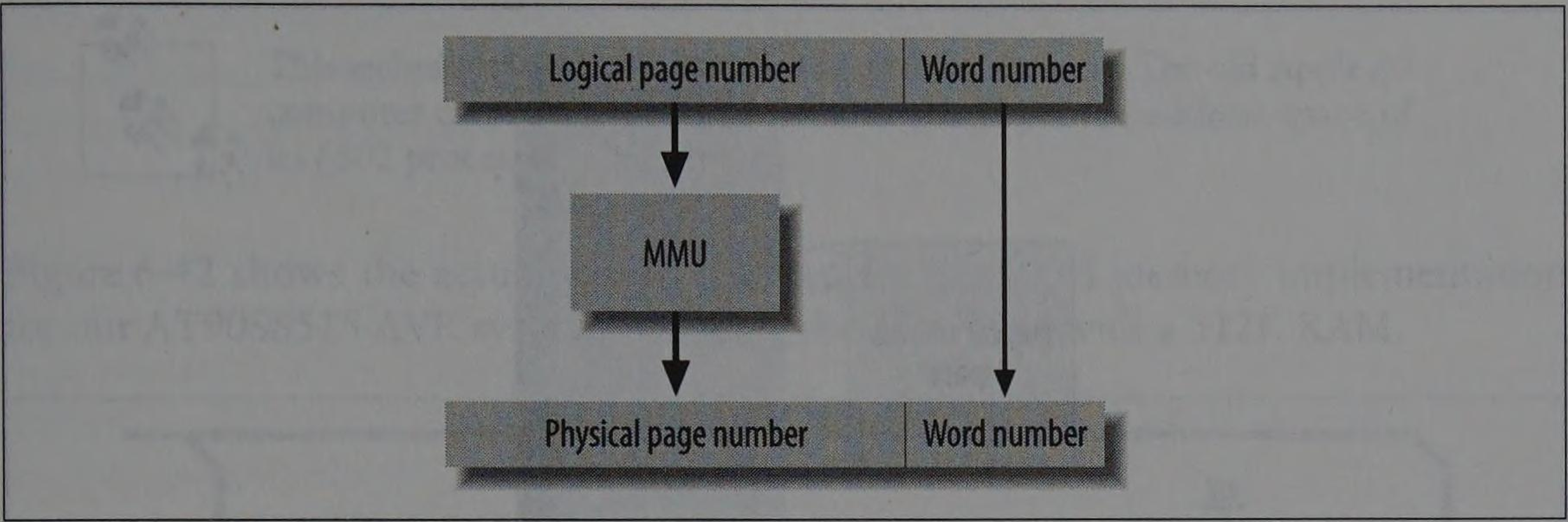


Figure 6-36. Address translation

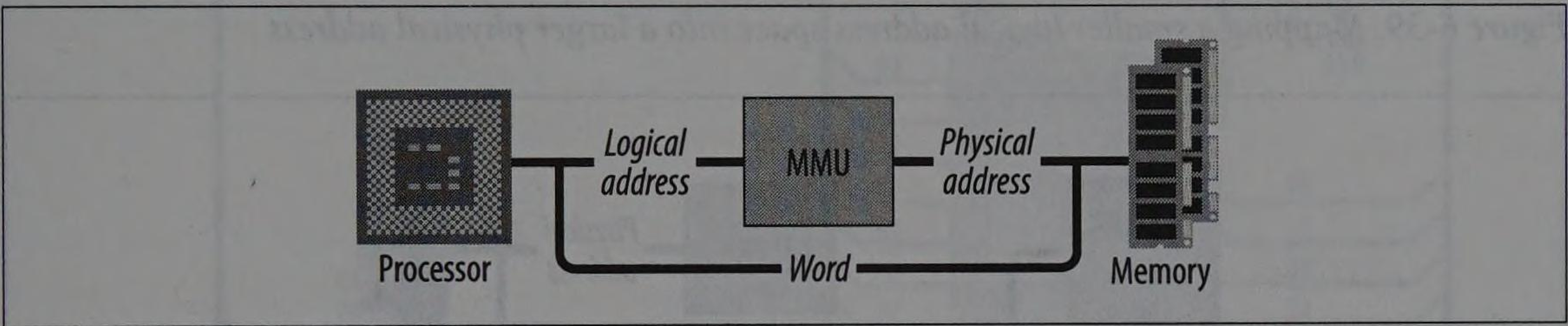


Figure 6-37. System using page address translation

Banked memory

The simplest form of memory management is when the logical address space is smaller than the physical address space. If the system is designed so that the size of each page is equal to the logical address space, then the MMU provides the page number, thus mapping the logical address into the physical address (Figure 6-38).

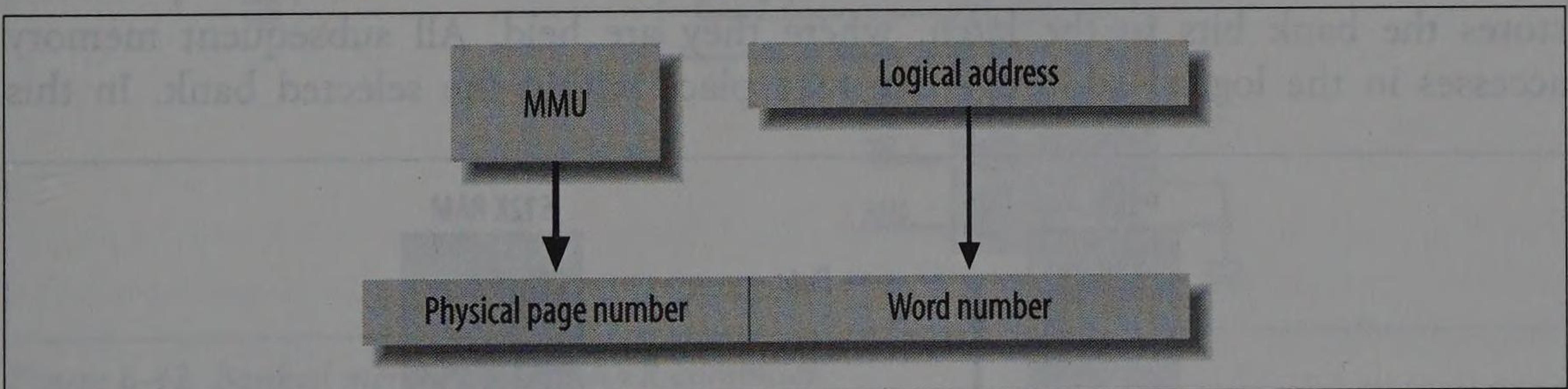


Figure 6-38. MMU generation of page number

The effective address space from this implementation is shown in Figure 6-39. The logical address space can be mapped (and remapped) to anywhere in the physical address space.

The system configuration for this is shown in Figure 6-40. This technique is often used in processors with 16-bit addresses (64K logical space) to give them access to larger memory spaces.

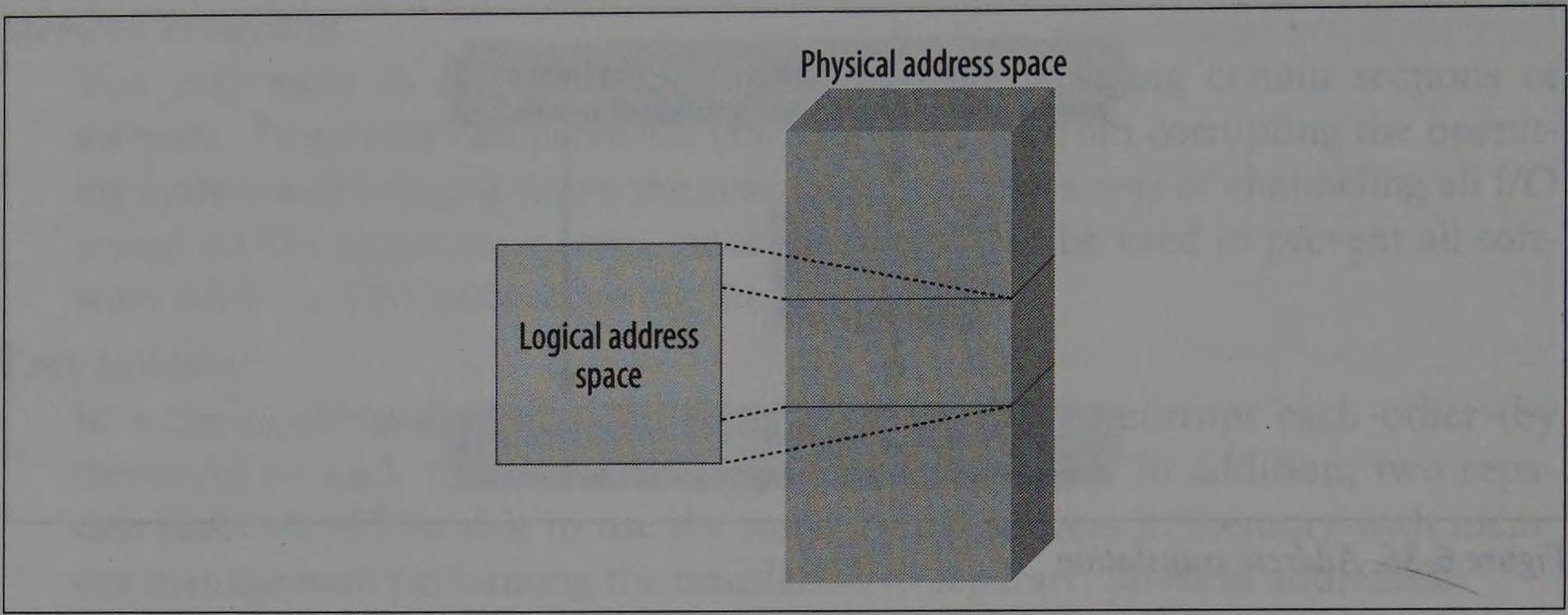


Figure 6-39. Mapping a smaller logical address space into a larger physical address

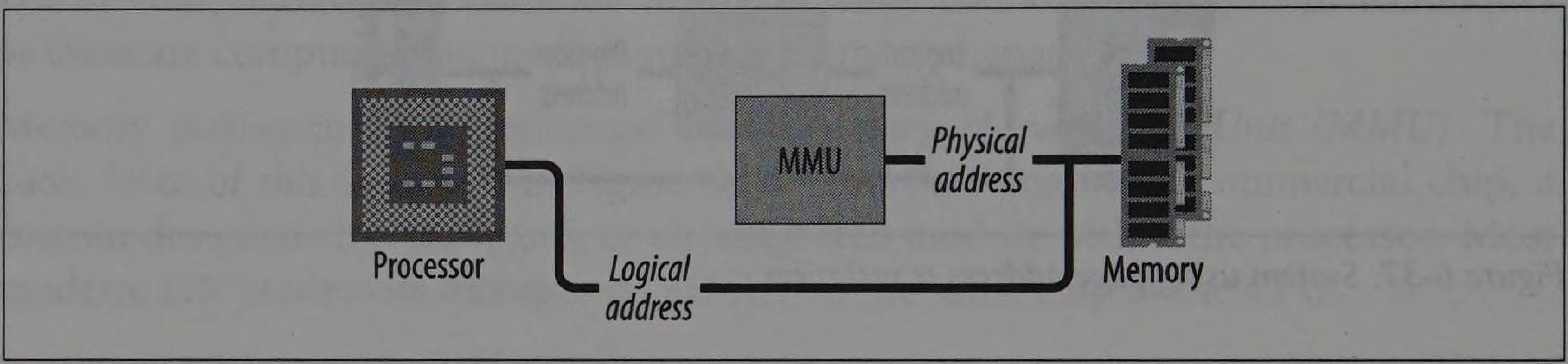


Figure 6-40. Generating a larger physical address

For many small systems, banked memory may be implemented simply by latching (acquiring and holding) the data bus and using this as the additional address bits for the physical memory (Figure 6-41). The latch appears in the processor's logical space as just another I/O device. To select the appropriate bank of memory, the processor stores the bank bits to the latch, where they are held. All subsequent memory accesses in the logical address space take place within the selected bank. In this

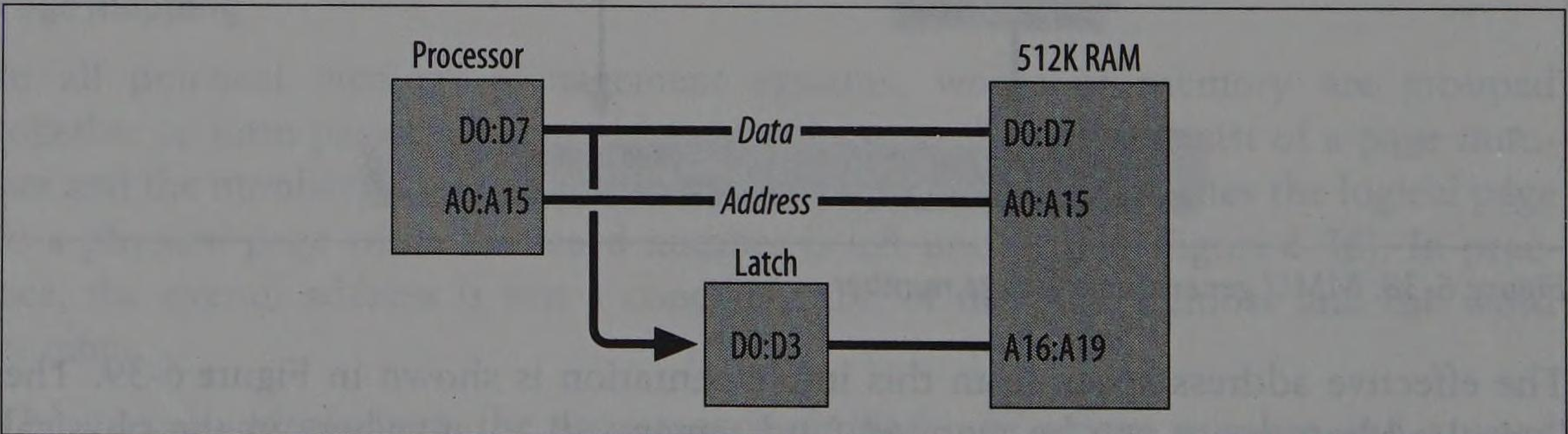


Figure 6-41. Simple banked memory implementation

example, the processor's address space acts as a 64K window into the larger RAM chip. As you can see, while memory management may seem complex, its actual implementation can be quite simple.

This technique has also been used in desktop systems. The old Apple /// computer came with up to 256K of memory, yet the address space of its 6502 processor was only 64K.

Figure 6-42 shows the actual wiring required for a banked memory implementation for our AT90S8515 AVR system, replacing the 32K RAM with a 512K RAM.

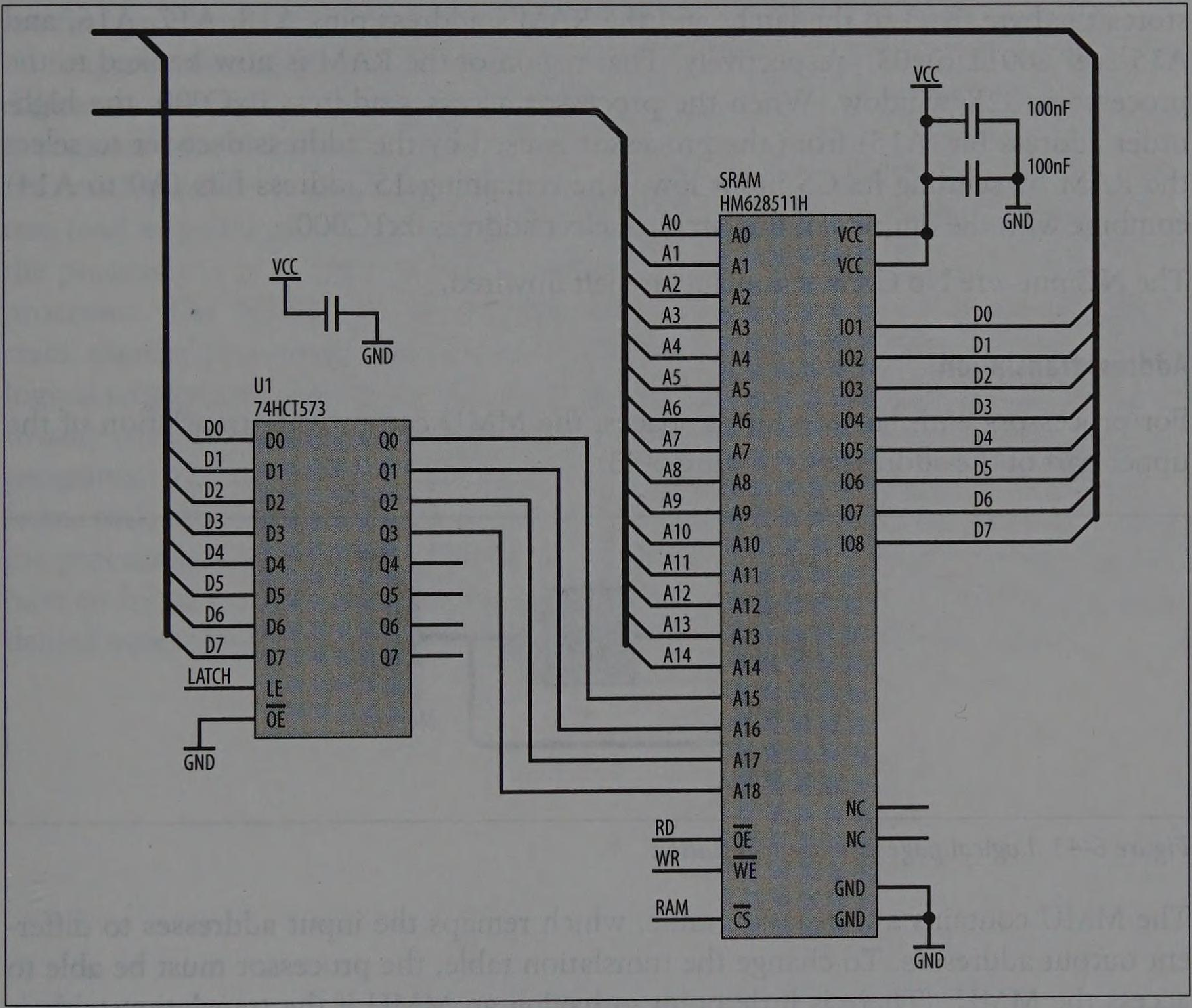


Figure 6-42. Banked memory for an AVR computer

The RAM used is an HM628511H made by Hitachi. In this implementation, we still have the RAM allocated into the upper 32K of the processor's address space as before. In other words, the upper 32K of the processor's address space is a window into the 512K RAM. The lower 32K of the processor's address space is used for I/O devices, as before. Address bits **A0** to **A14** connect to the RAM as before, and the data bus (**D0** to **D7**) connects to the data pins (**IO1** to **IO8***) of the SRAM.

* Memory chip manufacturers often label data pins as IO pins, since they perform data input and output for the device.

Now, we also have a 74HCT573 latch, which is mapped into the processor's address space, just as we did with the LED's latch. The processor can write to this latch, and it will hold the written data on its outputs. The lower nybble of this latch is used to provide the high-order address bits for the RAM.

Let's say the processor wants to access address 0x1C000. In binary, this is %001 1100 0000 0000 0000. The lower 15 address bits (**A0** to **A14**) are provided directly by the processor. The remaining address bits must be latched. So, the processor first stores the byte 0x03 to the latch, and the RAM's address pins **A18**, **A17**, **A16**, and **A15** see %0011 (0x03), respectively. That region of the RAM is now banked to the processor's 32K window. When the processor accesses address 0xC000, the high-order address bit (**A15**) from the processor is used by the address decoder to select the RAM by sending its $\overline{\text{CS}}$ input low. The remaining 15 address bits (**A0** to **A14**) combine with the outputs of the latch to select address 0x1C000.

The **NC** pins are No Connection and are left unwired.

Address translation

For processors with larger address spaces, the MMU can provide translation of the upper part of the address bus (Figure 6-43).

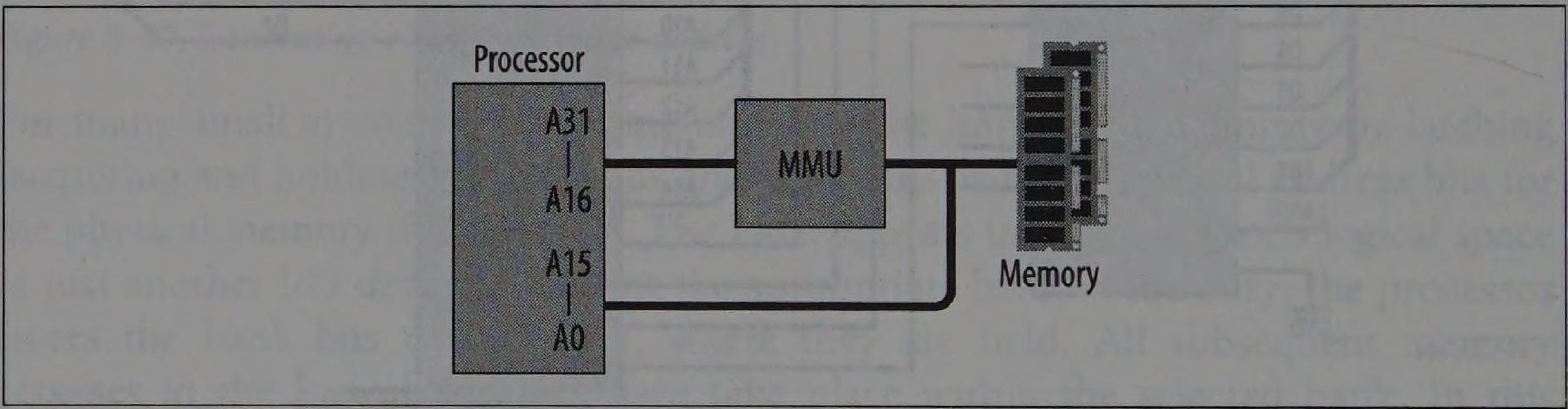


Figure 6-43. Logical page number translation

The MMU contains a translation table, which remaps the input addresses to different output addresses. To change the translation table, the processor must be able to access the MMU. (There is little point in having an MMU if the translation table is unalterable.) Some processors are specifically designed to work with an external MMU, while other processors have MMUs incorporated. However, if the processor being used was not designed for use with an MMU, it will have no special support. The processor must therefore communicate with the MMU as though it were any other peripheral device using standard read/write cycles. This means that the MMU must appear in the processor's address. It may seem that the simplest solution is to map the MMU into the physical address space of the system. In real terms, this is not practical. If the MMU is ever (intentionally or accidentally) mapped out of the

current logical address space (i.e., the physical page on which the MMU is located is not part of the current logical address space), the MMU cannot ever be accessed again. This may also happen when the system powers up, for the contents of the MMU's translation table may be unknown.

The solution is to decode the chip select for the MMU directly from the logical address bus of the processor. Hence, the MMU will lie at a constant address in the logical space. This removes the possibility of "losing" the MMU but introduces another problem. Since the MMU now lies directly in the logical address space, it is no longer protected from accidental tampering (by a crashing program) or illegal and deliberate tampering in a multitasking system. To solve this problem, many larger processors have two states of operation, a supervisor state and a user state with separate stack pointers for each mode. This provides a barrier between the operating system (and its privileges) and the other tasks running on the system. The state in which the processor is in is made available to the MMU through special status pins on the processor. The MMU may be modified only when the processor is in supervisor state, thereby preventing modification by user programs. The MMU uses a different logical-to-physical translation table for each state. The supervisor translation table is usually configured on system initialization, then remains unchanged. User tasks (user programs) normally run in user mode, whereas the operating system (which performs task swapping and handles I/O) runs in supervisor mode. Interrupts also place the processor in supervisor mode, so that the vector table and service routines do not have to be part of the user's logical address space. While in user state, tasks may be denied access by the operating system to particular pages of physical memory.

CHAPTER 7

68000-Series Computers

All is flux, nothing stays still.
—Heraclitus
Diogenes Laertius' Lives of Eminent Philosophers

This chapter examines the Motorola 68000, a 32-bit processor that has been around for quite some time and has evolved into a plethora of controllers and embedded processors. The 68000 (also known as the *68k*) is produced by Motorola (*http://e-www.motorola.com*) and is licensed by several other manufacturers. The range of 68000-based processors is large (check out the Motorola web site for a list of processors and their features). The number of applications that the 68000 has found its way into is enormous. You can even get 68000s as soft cores for FPGAs, which means that you place a 68000 CPU in the midst of your programmable logic, all on the one chip.

The 68000-series of processors are good general-purpose processors. They have a nice instruction set, are easy (and fun) to write code for, and are relatively easy to build computers around. They have large address spaces and asynchronous operation, allowing them to be interfaced to a wide variety of memory and peripherals of varying operating speeds. They are used in industrial control and monitoring and also in consumer electronics.

In this chapter, I look at the standard 68000 processor. More than likely, this is not the processor you will use in a design. Instead, you will probably choose a 68000-based integrated controller that better suits your needs. So, why look at a standard 68000 and not one of the derivatives? First, there are far too many diverse 68000-based processors to cover. Second, since all are based upon the 68000, understanding the basic 68000 is a great starting point. Finally, all the derivatives are generally easier to use than the original, so if you can design around a standard 68000, then you can design for a derivative processor as well.

The Motorola MC68000 was introduced in 1979 as the successor to the 8-bit 6800 family. It featured a large address space, 32-bit registers, a large number of addressing modes, and an enlarged instruction set with more than a thousand opcodes. It

was designed with the intention of running multitasking operating systems in general and specifically Unix. Its use in Unix machines has now long since passed, having been usurped by more advanced RISC processors. The 68000 processor was also used in the original Macintosh computers, as well as the Atari ST, the Commodore Amiga, and Jef Raskin's CAT computer,* all long extinct. Because of the processor's wide range of software and reasonable computing power, it is now used extensively in embedded systems. It now forms the basis of a family of microcontrollers designed for embedded systems, industrial control, networking, and PDAs. The 683xx series is the primary family of microcontrollers specifically tailored to embedded applications. These processors combine a CPU32 core (68020-based) with various integrated functions (such as UARTs, SPI, ADCs, etc.). Additional 68000 processors have been developed for specialized applications. The Palm PDA has a 68EZ328 DragonBall processor, also based on a CPU32 core, that incorporates an LCD controller along with many of the common functions found in PDAs. The DragonBall is essentially a PDA on a chip—just add memory. The uCLinux fraternity uses a DragonBall processor in its small embedded controller board.

The 68000 architecture was upgraded to RISC with the ColdFire series of processors. These see extensive use in industrial control and network interfaces.

Understanding the 68000 gives you access to a wide range of available processors. Dozens of commercial C compilers and assemblers are available for the 68000 family, as are a number of public-domain compilers. The 68000 is fully supported by the gnu development suite. Both Linux and BSD are also available for the 68000, as are numerous commercial operating systems.

The 68000 Architecture

The 68000 has eight 32-bit data registers (D0–D7), eight 32-bit address registers (A0–A7), a 32-bit program counter, two 32-bit stack pointers, and a 16-bit status register (Figure 7-1). The processor is capable of handling data as either 32-bit-long words, 16-bit words, bytes, or bits.

The processor has two modes of operation, supervisor mode (operating system) and user mode (applications). The mode of operation is made available to external hardware, thereby allowing the address decoder to have separate supervisor and user spaces.

The standard 68000 is just a conventional bus-based processor. A block diagram of a generic 68000-series processor is shown in Figure 7-2. The figure also shows the pins for an example 68000-series processor. The pins and signals of 68000s can vary from one device to another, but they all have the same core functionality. The embedded controllers add to this basic functionality with additional I/O capability. We'll look

* For an interesting overview of the CAT, read Jef Raskin's *The Humane Interface*. He discusses the CAT's unique design and has some interesting ideas on user interface design.

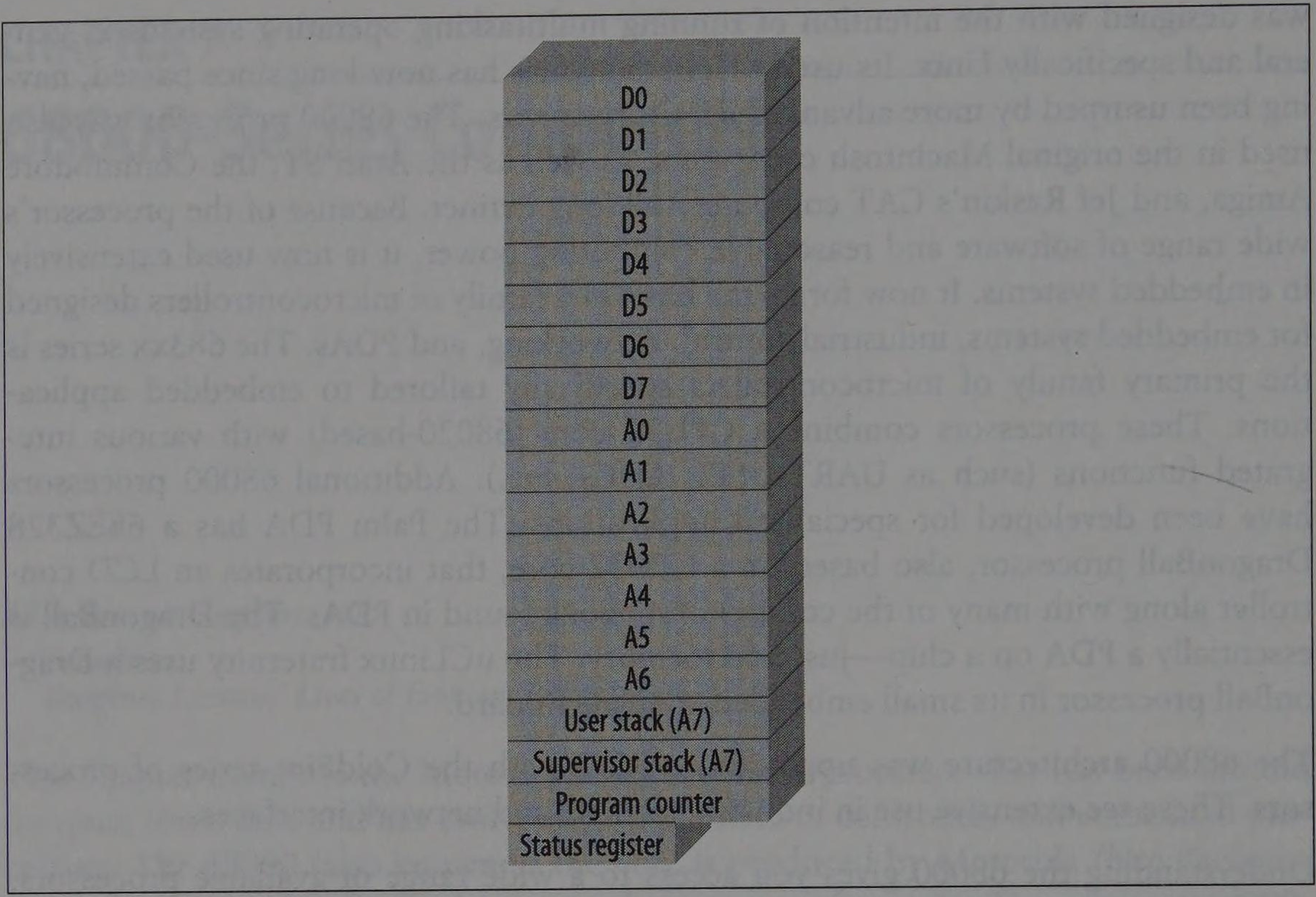


Figure 7-1. Programmer's model of the 68000

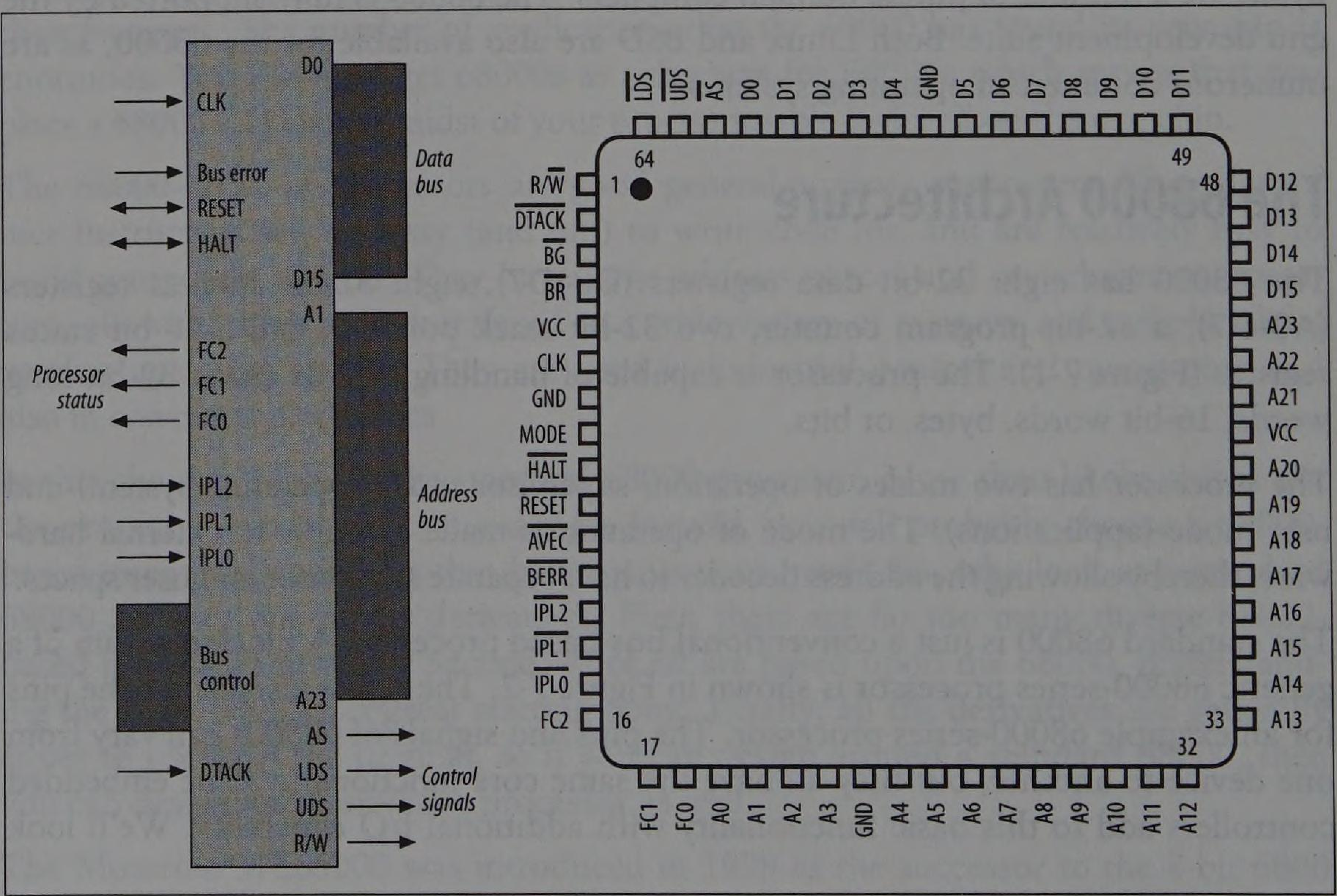


Figure 7-2. MC68000 block diagram and pinout

at the pins for the MC68EC000 shortly. You can download datasheets, programmers' manuals, and technical references for 68000-series processors from the Motorola semiconductor web site, *http://e-www.motorola.com*. While you're there, check out the other Motorola processor families, which range from tiny 8-bit controllers to 64-bit PowerPC RISC processors, and everything in between.

The original 68000 has a 23-bit address bus (A1 to A23), giving it access to a memory space of 16M, and a 16-bit data bus. Most other processors based on the 68000 architecture have address and data buses of 32 bits and can therefore access up to 4G of memory.

The processors have an input clock that drives all processor operation. Memory accesses typically take eight input clock cycles, provided that wait states are not introduced. Many processors based upon the 68000 incorporate built-in address decoding and software-configurable wait-state generation, making interfacing much simpler.

The processors have an address strobe (**$\overline{\mathrm{AS}}$**) indicating when a valid address is present on the bus, data strobes (**$\overline{\mathrm{LDS}}$**, **$\overline{\mathrm{UDS}}$**) indicating valid data, and an **$\mathrm{R}/\overline{\mathrm{W}}$** line that shows the direction of the transfer. In addition, a Data Transfer Acknowledge input, **$\overline{\mathrm{DTACK}}$**, is used by external devices to indicate to the processor that it may terminate its current memory cycle. (Some 68000 processors call their Data Transfer Acknowledge **$\overline{\mathrm{DTACKB}}$**.) The function code outputs (**FC0**, **FC1**, and **FC2**) indicate the current operating mode (supervisor* or user) of the processor. Bus Error (**$\overline{\mathrm{BERR}}$**) is used by an external address decoder to indicate an error condition. This allows the system to trap out accesses to unused regions of memory space or, in combination with the status lines, to detect user access to memory space allocated for supervisor use only. For example, if a program crashes and in the process of crashing attempts to access a region of memory to which no device is allocated, the address decoder is able to signal that fault back to the processor. An assertion of **$\overline{\mathrm{BERR}}$** causes the processor to execute an interrupt and take appropriate action. **$\overline{\mathrm{HALT}}$** is used to suspend processor operation without generating a reset. Three interrupt inputs (**$\overline{\mathrm{IPL0}}$**, **$\overline{\mathrm{IPL1}}$**, and **$\overline{\mathrm{IPL2}}$**) are used to generate seven levels of external interrupt handling. Bus Grant (**$\overline{\mathrm{BG}}$**) and Bus Request (**$\overline{\mathrm{BR}}$**) are DMA control signals by which another processor can arbitrate to acquire the computer's buses. The **MODE** pin, present on only some 68000 processors, determines whether the 68000 uses its data bus as 16 bits or 8 bits. **MODE** is sampled as the processor comes out of reset. **$\overline{\mathrm{AVEC}}$**, also found in only some 68000 processors, determines whether the processor uses autovectoring for its interrupts. If autovectoring is enabled, the processor will expect the interrupting peripheral to supply the appropriate vector. This allows a peripheral to specify what type of action the processor needs to take when a given interrupt is generated. Other 68000 processors may have other signals as well, but these are the main ones.

* In multitasking, multiuser systems, the operating system runs in supervisor mode while user applications (and their data) are accessed in user mode.

The basic timing diagram for a 68000 memory access is shown in Figure 7-3.

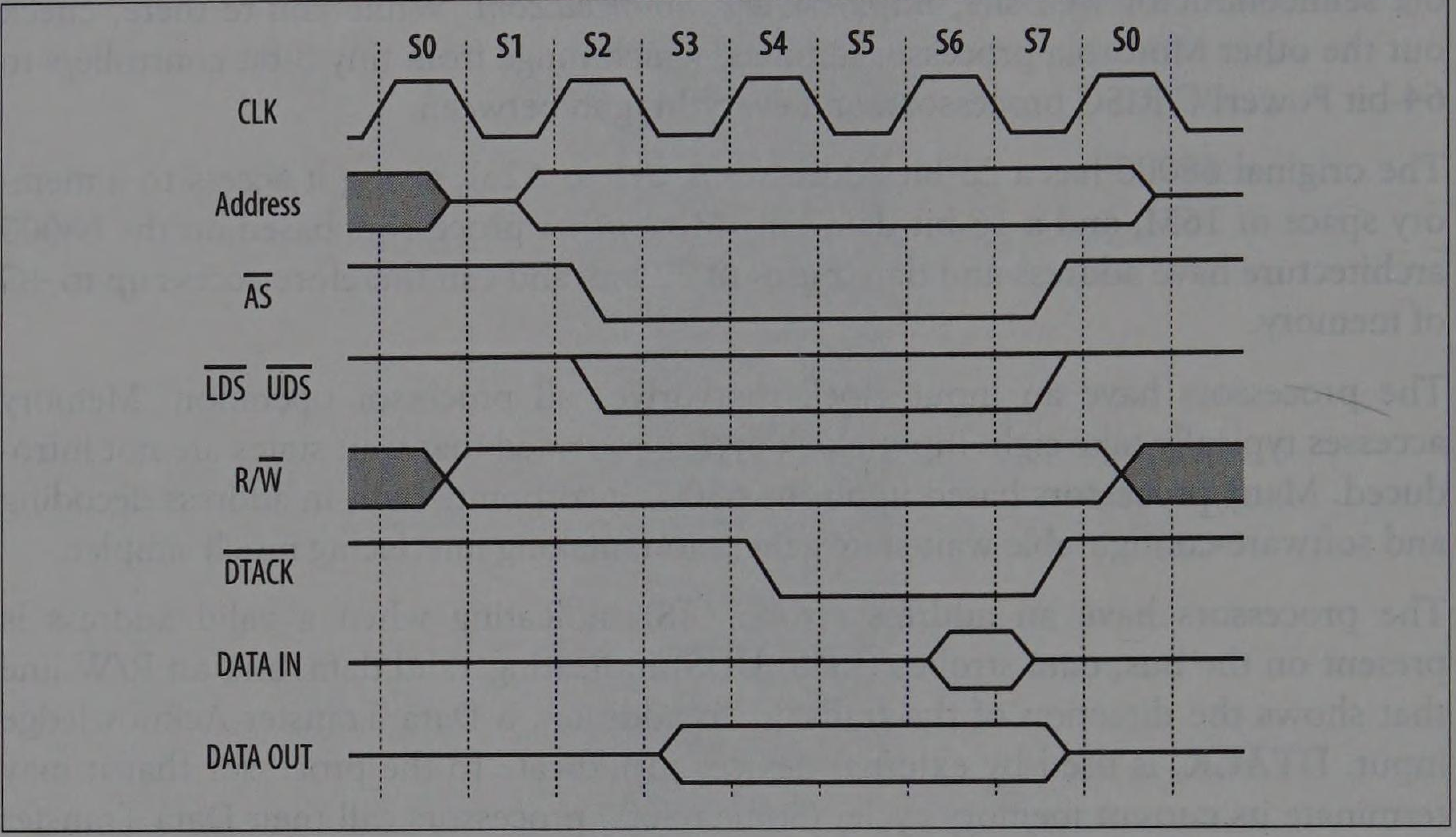


Figure 7-3. MC68000 timing diagram

The memory cycle of a 68000 is divided into a number of clock states, S0 through to S7. The cycle begins with state S0. The processor validates **R/$\overline{\textbf{W}}$** for the coming cycle, sending it low for a write access, driving it high for a read access. The processor also tristates its address bus from the previous memory access. By S2 the processor has output a valid address and drives the address strobe ($\overline{\textbf{AS}}$) low indicating that a valid address is present. The lower and upper data strobes ($\overline{\textbf{LDS}}$ and $\overline{\textbf{UDS}}$) go low as appropriate and indicate the width of the memory access taking place. For a 16-bit transfer, both $\overline{\textbf{LDS}}$ and $\overline{\textbf{UDS}}$ assert. For an 8-bit transfer, only one of $\overline{\textbf{LDS}}$ or $\overline{\textbf{UDS}}$ will assert, depending on whether the upper byte or lower byte is being transferred. If the current memory access is a write cycle, the processor outputs valid data in state S3. At this point, all outputs from the processor are now valid and the processor waits for the device being accessed to respond.

At the falling edge of the clock in S4, the processor begins checking the state of the Data Transfer Acknowledge ($\overline{\textbf{DTACK}}$) input. If $\overline{\textbf{DTACK}}$ is high, the processor inserts wait states and continues to do so until $\overline{\textbf{DTACK}}$ is found to be low on the falling edge of the clock. (I'll discuss how to generate wait states later in the chapter.) When $\overline{\textbf{DTACK}}$ is low, the processor recognizes this as an indication that the device being accessed has had sufficient time to respond and prepares to terminate the cycle. If the cycle is a read cycle, the processor will latch data on the falling edge of the clock in state S6. If it is a write cycle, the device being accessed will latch data as the data strobes go high in S7.

Support for synchronous operation is also provided for, using control signals found in the old 6800 series of processors. Since 6800s have long since passed into history and 6800-compatible peripherals are now exceptionally rare, just ignore the 6800 control signals. Most 68000-based derivative processors no longer include support for 6800 peripherals.

A Simple 68000-Based Computer

Let's look now at a small 68000-based computer. For simplicity, we'll give it just a small amount of memory and a single peripheral, an MK68901 *MFP* (*Multifunction Peripheral*) produced by ST Electronics. The MFP gives us a UART (covered in detail in Chapter 10), parallel I/O, and interrupt control. A block diagram of the system is shown in Figure 7-4.

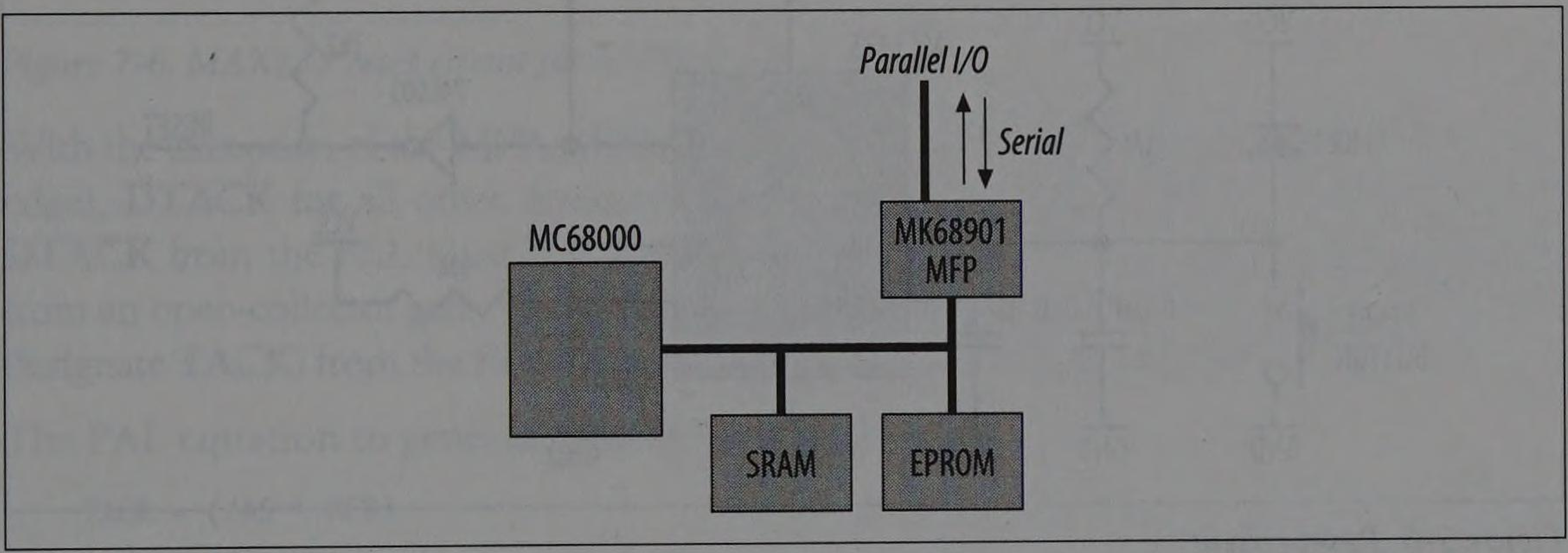


Figure 7-4. A 68000-based computer

This system is designed with only a small amount of memory, so as to keep the design uncomplicated. While this is not much compared to many desktop machines, it is sufficient for many small control applications.

This design could be used for a number of simple applications. The counters of the MK68901 may be used to monitor external event pulses or to generate PWM for motor control. (We'll see how to do that in Chapter 12.) This computer could also be used to accept commands through its serial port and activate (or deactivate) external subsystems using the parallel I/O pins of the MK68901. This basic design could also be adapted to provide a bridge between an RS-232C interface (Chapter 10) and a parallel port. You could use this to interface a parallel-port printer to a serial-port-only computer. Alternatively, you could use it to put a serial modem on your PC's parallel port. Using the bus-interfacing techniques we learned in Chapter 5, you could add additional peripherals such as ADCs and DACs (Chapter 12), Ethernet (Chapter 11), or a whole range of other devices. The list of possible applications is endless. And it all starts with this core design.

So, let's start our tour of a 68000-based computer system. We'll look at the reset circuit, address decoder, I/O, and memory, in turn.

Reset Circuit

To reset an MC68000, both $\overline{\textbf{RESET}}$ and $\overline{\textbf{HALT}}$ must be driven low simultaneously. In addition, both of these signal lines may also act as outputs from the processor. Therefore, both must be independently driven by the reset circuit through open-collector gates. The conventional way of doing a 68000 reset circuit is shown in Figure 7-5.

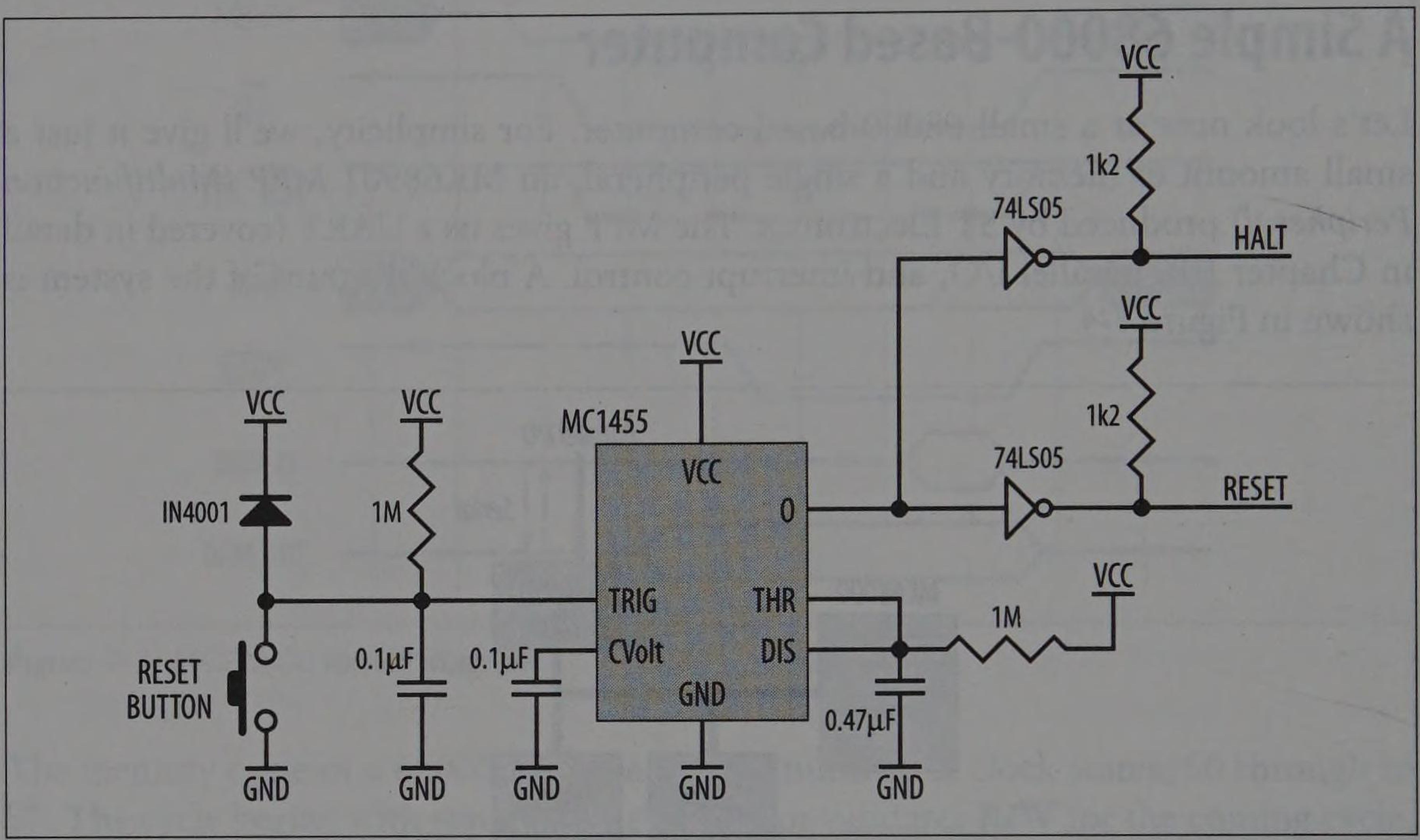


Figure 7-5. Reset circuit

The MC1455 will respond to a disruption on V_{CC} by sending its output low. This output is used to drive $\overline{\textbf{RESET}}$ and $\overline{\textbf{HALT}}$ low simultaneously. In normal operation, $\overline{\textbf{RESET}}$ is held high by the pull-up resistor, unless pulled low through the reset switch being pressed. The diode is present to remove any glitches that might send **TRIG** above V_{CC}.

A better reset circuit is shown in Figure 7-6, using a MAX825 integrated reset controller. Again, both $\overline{\textbf{RESET}}$ and $\overline{\textbf{HALT}}$ need to be driven low.

Address Decoder

Logic to perform address decoding and the generation of separate read and write strobes is implemented in a PAL. (We covered PALs and PAL equations in Chapter 6.) In each case, $\overline{\text{AS}}$ (Address Strobe) of the processor is used as an indication of a valid address present on the bus. The address decode equations are as follows:

```
ROM = /(/AS * /A23 * /A22)
RAM0 = /(/AS * /A23 * A22 * /LDS)
RAM1 = /(/AS * /A23 * A22 * /UDS)
MFP = /(/AS * A23 * /A22)
```

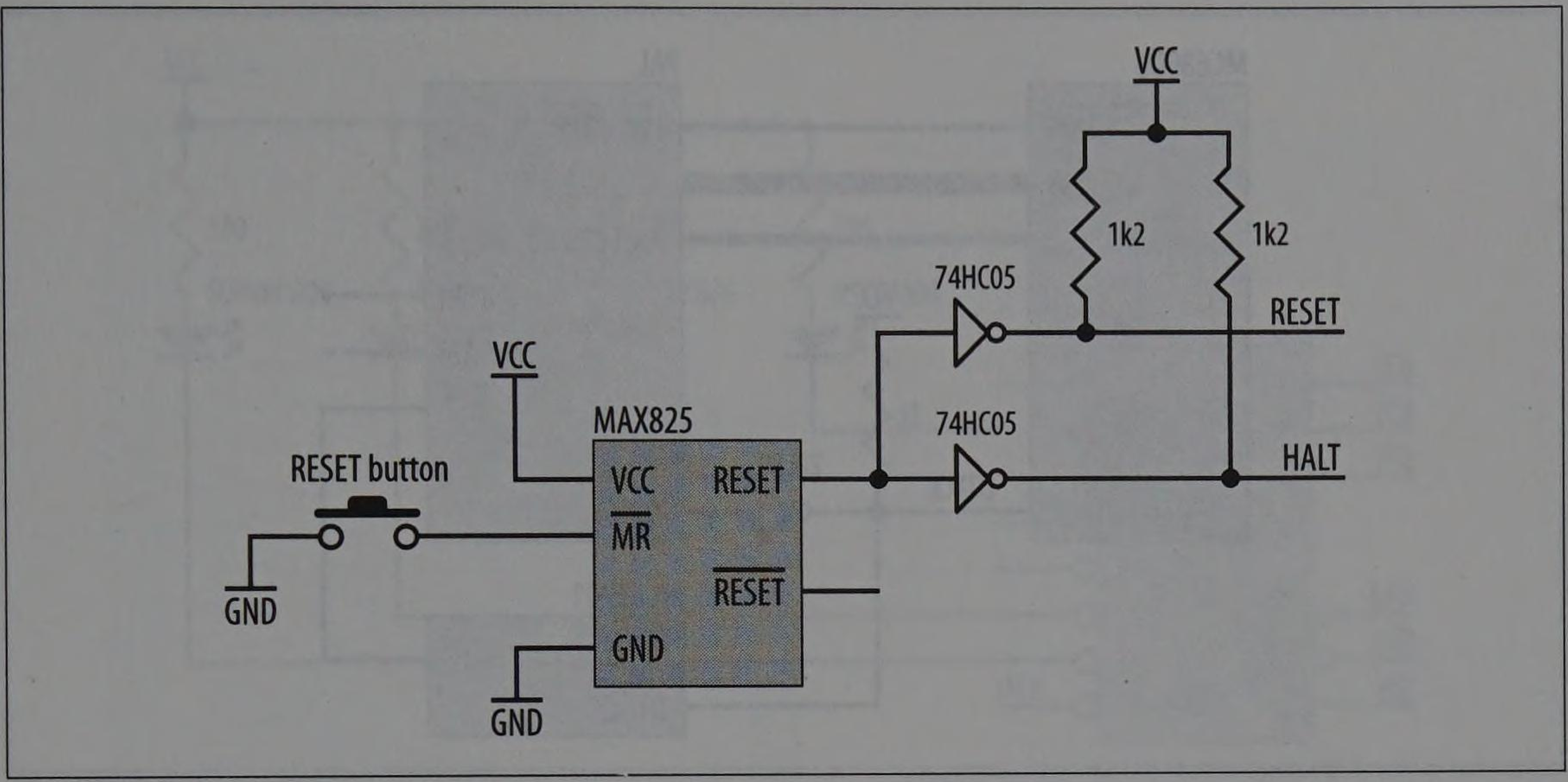


Figure 7-6. MAX825 reset circuit for a 68000

With the exception of the MFP, which generates its own **DTACK** (Data Transfer Acknowledge), **DTACK** for all other devices is generated as part of the address decoding. Since **DTACK** from the PAL must be OR-tied with **DTACK** from the MFP, it must be driven from an open-collector gate. Therefore, we generate a high-active acknowledge (which we'll designate **TACK**) from the PAL and invert this through an open-collector 74LS05.

The PAL equation to generate **TACK** is simply:

```
TACK = (/AS * MFP)
```

Therefore, **TACK** is active (high) whenever the processor accesses its address space, so long as it is not accessing the MFP. If the address strobe is high, or if there is an access to the MFP, then **TACK** is low. The **TACK** output from the PAL is inverted through an open-collector 74LS05 and OR-tied with **DTACK** from the MFP. **DTACK** requires a pull-up 1kΩ resistor, since this input must have a sharp rise time. A block diagram is shown in Figure 7-7.

No provision for generating a **BERR** (Bus Error) is made because our simple address decoding allocates all of the address space. If we had any unused regions of the memory space, we would use our address decoder to generate a **BERR** when accesses to the unused regions were made.

The PAL equations to generate separate read and write strobes for the memory chips are:

```
UWE = /(/UDS * RW)
LWE = /(/LDS * RW)
UOE = /(/UDS * /RW)
LOE = /(/LDS * /RW)
```

The connections for the PAL are shown in Figure 7-8. Additional addresses are brought into the PAL to allow for future changes to the memory map. The processor's clock (**CLK**) is used by the PAL to generate the clock for the MFP (**MFPCLK**).

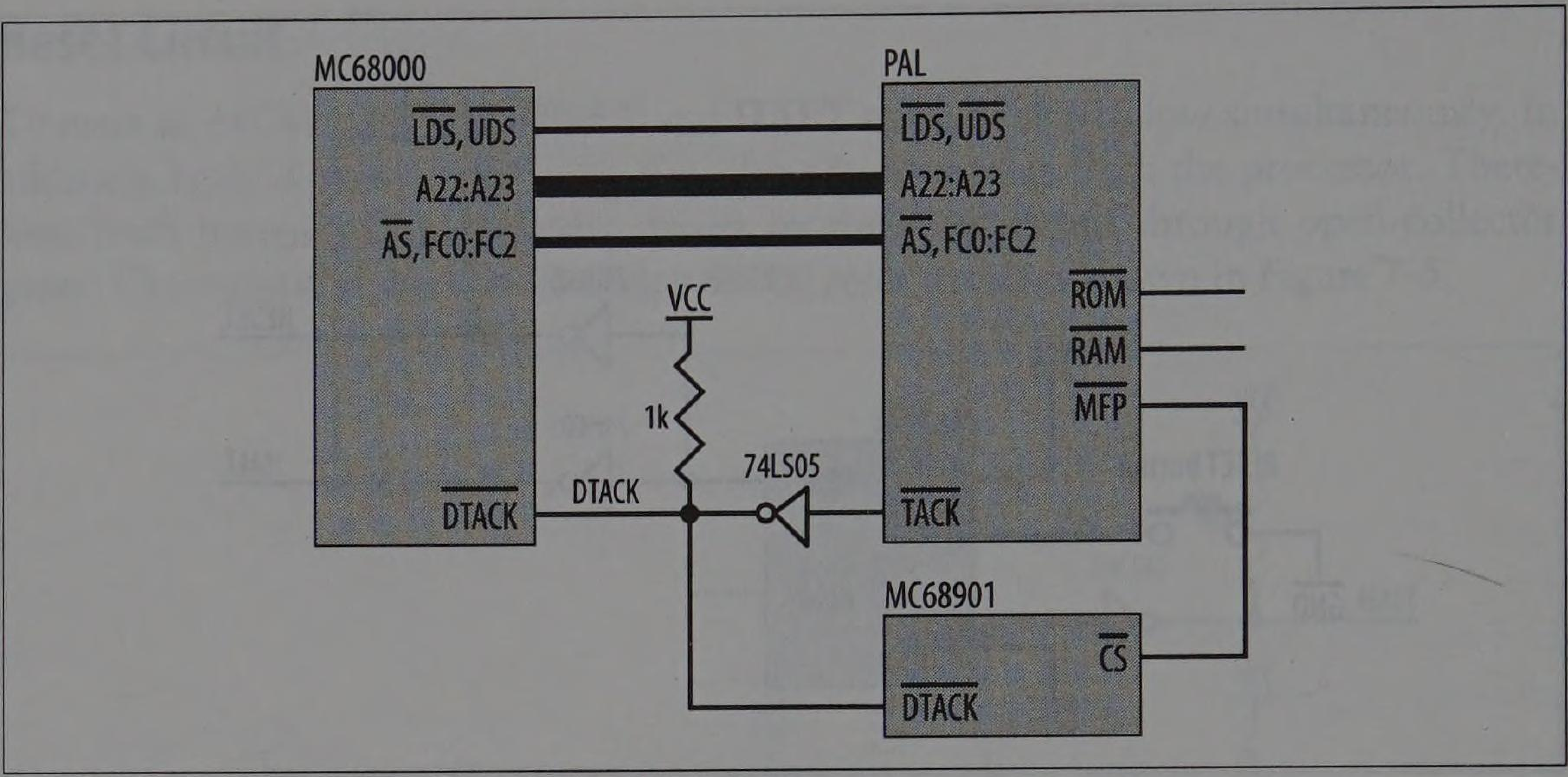


Figure 7-7. Address decode and **DTACK** *generation*

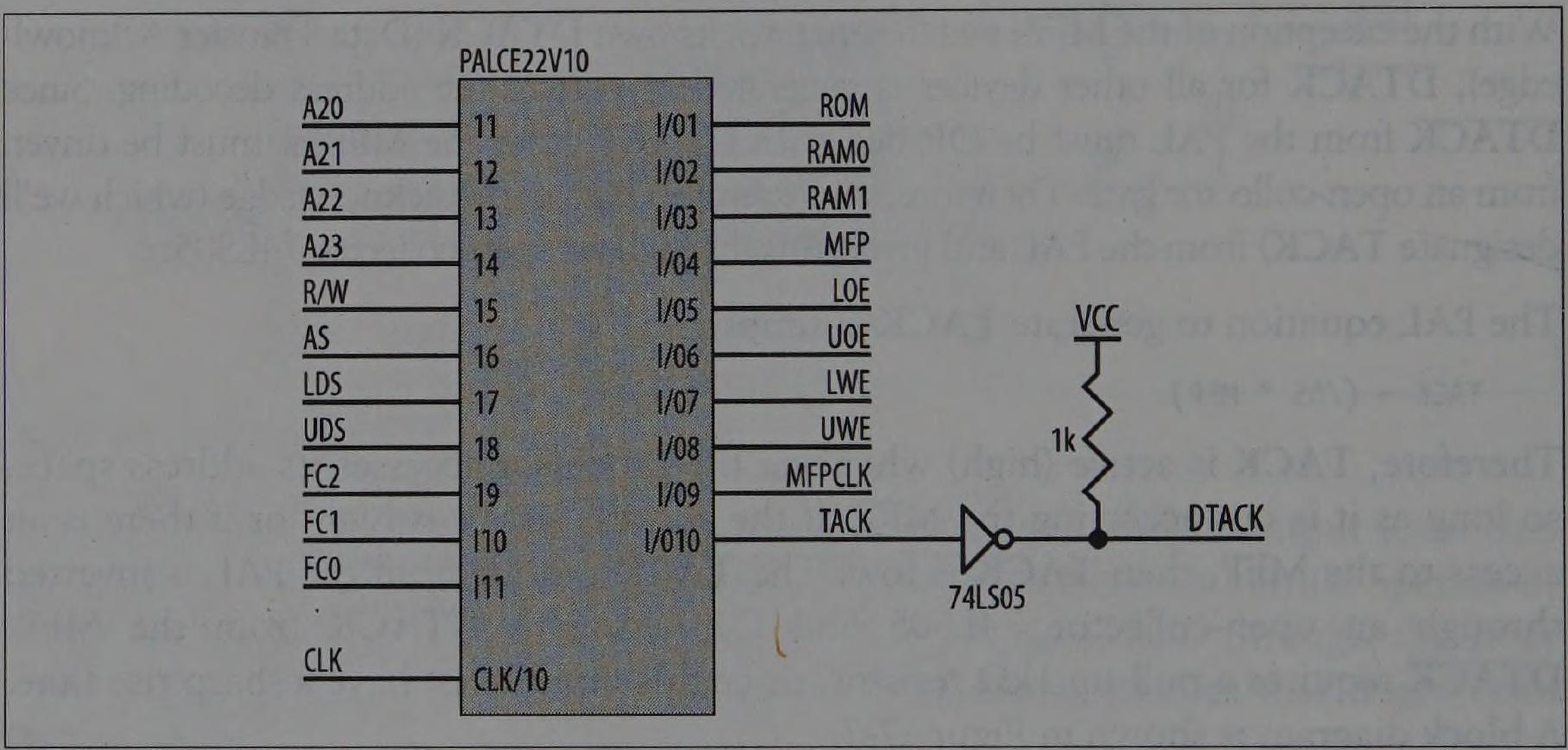


Figure 7-8. Address decode and system logic PAL

The function code outputs (**FC0–FC2**) can be decoded using a 74LS138 to drive three LEDs (Figure 7-9). These provide a visible indication of processor status. The function codes could also be used by the address decoder if you wanted to have separate user and supervisor address spaces. Many of the more sophisticated peripheral chips (such as the MFP) require the processor to acknowledge when they have generated an interrupt. The 74LS138 also uses the function codes to generate an Interrupt Acknowledge (**IACK**) for peripherals, since the function codes also indicate an IACK condition.

I/O

The MK68901 MFP provides a serial port as well as basic parallel I/O functions, 16-source interrupt controller, and four 8-bit timers. (Serial ports are covered in

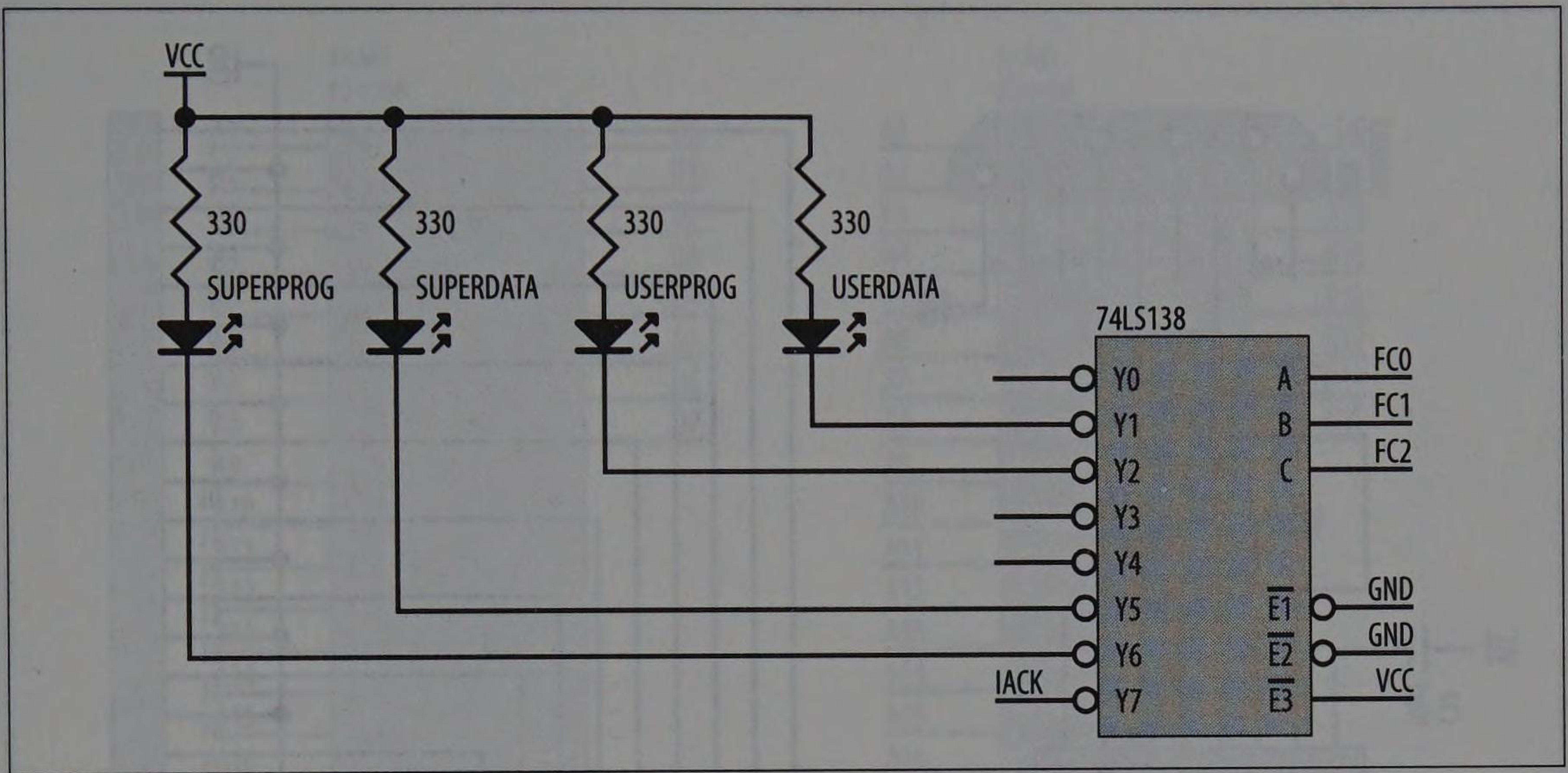


Figure 7-9. Status LEDs indicating processor mode

detail in Chapter 10.) The MK68901 has an internal oscillator that drives the internal timers. A timer output (**TDO**) is fed back into the MFP as the clock for the serial interface. The internal oscillator must therefore run at a frequency appropriate for RS-232C. Thus, the oscillator is controlled by an external 3.6864MHz crystal, which can be divided down by the MFP to provide the appropriate baud rates for the serial port. The serial lines from the MFP are converted to RS-232C voltage levels by a MAX3232 level shifter. A nine-pin, D-type connector provides access to the RS-232C signals. The parallel I/O lines and timer inputs and outputs are also made available through a 26-pin IDC connector.

The schematic for the MFP is shown in Figure 7-10.

Memory

The system is designed with 256K of EPROM and 512K of static RAM. The connections to the SRAM are shown in Figure 7-11. Note that since the data bus of a 68000 is 16-bits wide, two SRAMs are required. For 68000-based derivatives with 32-bit external data buses, four memory chips would be required in parallel. Note how half the data bus goes to one chip and the other half goes to the other chip.

Now, note the address lines going to the SRAMs. The lowest address bit from the processor is **A1**, and this is connected to the **A0** inputs of the SRAMs and so on. Since the processor accesses external memory in 16-bit words, **A1** represents the least-significant address bit. In other words, as you move from word to subsequent word in memory, it is **A1** that increments. **A0** is the least-significant address bit of the SRAMs, but since the two SRAMs together form a 16-bit word of memory, the **A0** of the SRAMs must connect to **A1** of the processor. The other address bits follow on from that starting point.

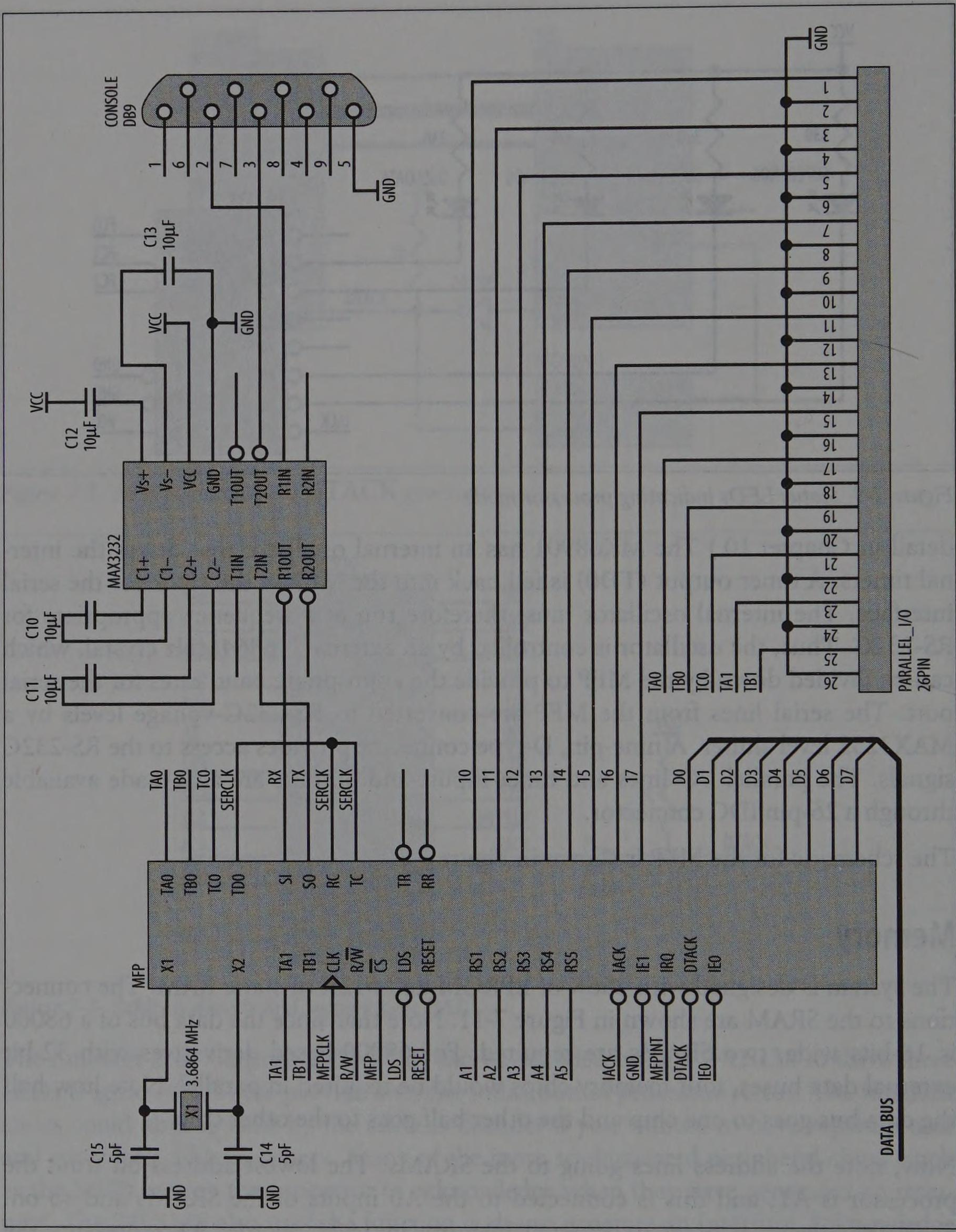


Figure 7-10. Multifunction Peripheral

Similarly, the connections for the ROMs are shown in Figure 7-12.

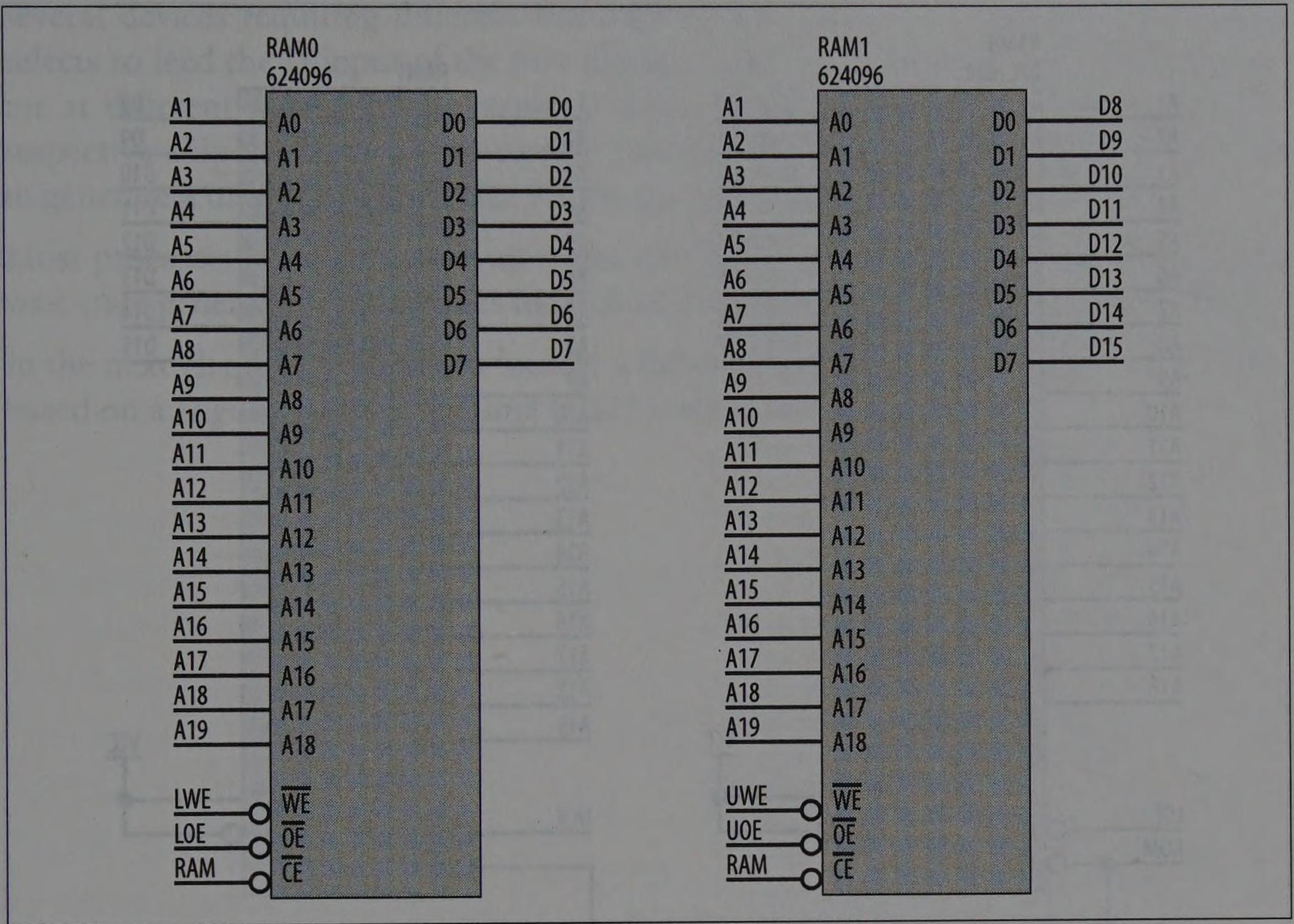


Figure 7-11. Interfacing to SRAM

Wait States

Depending on the speed of your processor, the access times of your memory, and your peripheral chips, you may need to introduce wait states into the 68000's memory cycle. Wait-state generation is basically the same principle for processors that support asynchronous memory cycles. The processor will have an input (sometimes more than one) that will cause it to delay the memory cycle, giving slower devices time to respond. In the case of the 68000, that input is $\overline{\textbf{DTACK}}$. To insert a wait state for a given device, we need to detect an access to that device and hold $\overline{\textbf{DTACK}}$ inactive for the required additional clock cycles. In other words, use the chip select for a given device to delay $\overline{\textbf{DTACK}}$ going low. The circuit to do this is simple and is best done inside a PAL or other programmable logic device. This facilitates changing the wait-state generator if faster parts are used in the design at a later stage. The wait state generator consists of a series of D-type flip-flops* (Figure 7-13). Each flip-flop represents an additional clock cycle that the transfer acknowledge is delayed.

* A flip-flop is a logic element that feeds the D input through to the Q output on the changing edge of a clock pulse.

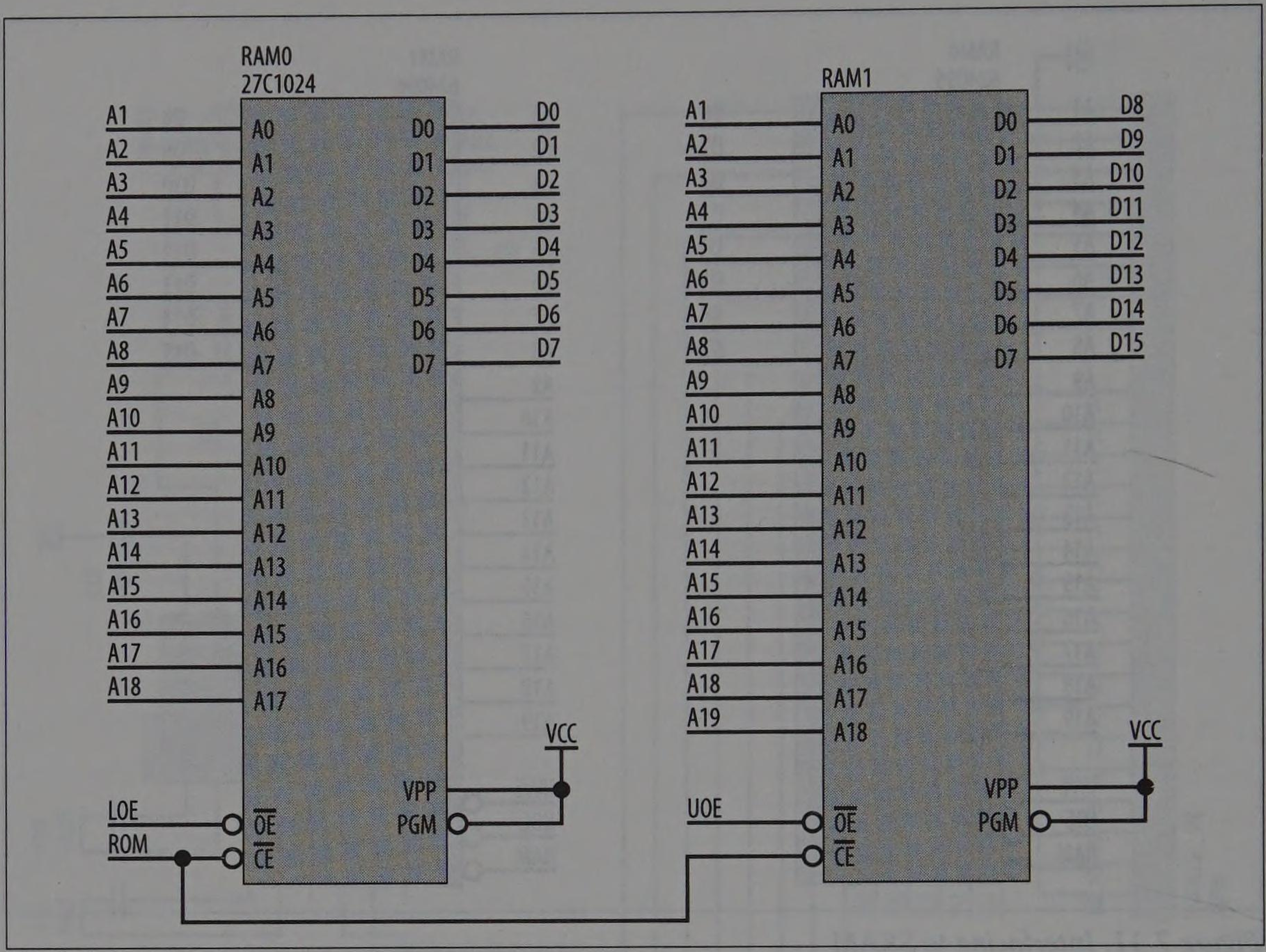


Figure 7-12. Interfacing to EPROMs

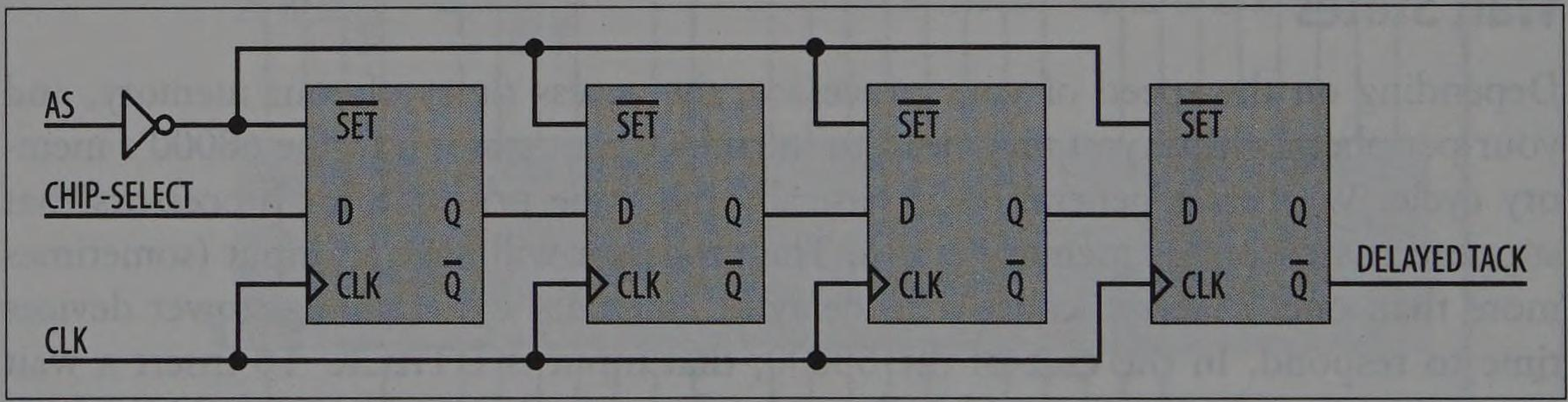


Figure 7-13. Wait-state generator

Between memory cycles, the address strobe, $\overline{\textbf{AS}}$, goes high. This is first inverted and then connected to the low-active $\overline{\textbf{SET}}$ input of each of the flip-flops. Thus, the output of each of the flip-flops is driven high between each memory cycle. This resets them from any previous cycle. The address decoder generates a chip select for the particular device, and this is connected to the D input of the first flip-flop. So, on each successive clock pulse, the 0 provided by the chip select is clocked through from one flip-flop to the next. After four clock pulses, the 0 has arrived at the Q output of the last flip-flop. The inverted output of this flip-flop, $\overline{\textbf{Q}}$, becomes a 1. This is then output by the PAL to be inverted by the 74LS05 open-collector inverter to provide $\overline{\textbf{DTACK}}$ for the processor. For additional wait states, add more flip-flops. For

several devices requiring different numbers of wait states, use their combined chip selects to feed the D input of the first flip-flop, then "tap" into the wait-state generator at different stages for the required delay. Each of these taps is gated with the respective chip select to enable/disable that output, before recombining all of them to generate a unified acknowledge for the processor.

Most processors that support wait states now include built-in, software-configurable wait-state generators. This makes the task of designing the system logic much simpler.

In the next chapter, we'll take a look at a different sort of embedded processor—one based on a *Digital Signal Processing* (*DSP*) architecture.

CHAPTER 8

DSP-Based Controllers

. . . the uniformity of the world, that everything which happens is connected, that the great and the small things are all encompassed by the same forces of time . . . this unity and necessary sequence of all things is nevertheless broken in one place, through a small gap, this world of unity is invaded by something alien, something new . . .

—Hermann Hesse
Siddhartha

This chapter takes a look at *Digital Signal Processors*, or *DSPs*, which are special-purpose processors designed for executing mathematically intensive algorithms. They first appeared in the early 1980s and since then have expanded into a wide range of devices used in a variety of applications. These processors are characterized by their ability to quickly move data in and out of memory (or a peripheral), and their architectures are optimized for mathematical processing of that data.

The basic purpose of a DSP is to rapidly read in some data, perform a complex algorithm on it, then move the result out. Many DSPs have dual data spaces, known as X and Y. They are able to access both data spaces simultaneously, retrieving two operands at once for processing. As well, many DSPs are also Harvard architecture, and so have three separate address spaces, one for code and two for data, all of which can be accessed concurrently. That ability, combined with very sophisticated ALUs, gives DSPs their advanced data-processing prowess.

DSPs are commonly used in audio processing, video or image processing, communications, radar and sonar systems, and biomedical applications. Your cell phone has a DSP in it. So does your DVD player and the surround-sound (AV) amplifier in your home theater system. The so-called bionic ear, made by Cochlear, uses a DSP.

Some example applications of DSPs are:

- Engine control and antiskid brakes in cars
- Digital radios and TVs

- DVD players and home theater systems
- Music synthesizers
- GPS navigation
- Radar and sonar processing
- Aircraft navigation and guidance, spacecraft avionics, missile guidance systems
- Industrial motor control
- Robotics
- Virtual-reality systems
- Image processing, compression, and enhancement
- Pattern recognition and machine vision
- Adaptive filtering, Fast Fourier Transforms (FFTs), Hilbert transforms
- Scientific data processing
- Medical diagnostic equipment, ultrasound, and medical imaging systems
- Cell phones, pagers, modems, cell phone base stations, digital fax machines
- Data encryption
- Digital PABXs, ADSL
- Echo cancellation
- Spread-spectrum processing in communications
- Videoconferencing systems
- Speaker verification
- Speech enhancement and recognition
- Speech synthesis and coding
- Voice mail systems

And that's just for starters.

The three big manufacturers of DSPs are Texas Instruments (*http://www.ti.com*) with the TMS320 series, Analog Devices (*http://www.analogdevices.com*) with the 21xx and SHARC (21xxx) processors, and Motorola (*http://e-www.motorola.com*) with the DSP56xxx processors and the high-end MSC8100 *StarCore* processors designed for communications and network processing. Many other manufacturers are starting to add DSP functionality into their embedded controllers. An example of this is the dsPIC processor by Microchip (*http://www.microchip.com*).

TI's DSPs range from small, low-cost units to supercomputers on a chip. The TMS320C6000 series makes your average PC look like a rusty abacus in comparison. They are 128-bit *VLIW* (*Very Long Instruction Word*) processors and can execute up to eight instructions every clock cycle. They can run at up to 2000MIPS and 900MFLOPS, and TI is working to make them even faster. These are processors designed for serious number crunching. (And if you want to play with one, you'll need serious dollars.)

Both the TI and Analog Devices DSPs are designed for use as building blocks in parallel DSP computers. The Analog Devices SHARC supports both message-passing MIMD and shared-memory MIMD in the one machine. You can have six SHARCs as a shared-memory parallel computing node, and you can have six of these nodes message passing with one another. When you consider that each SHARC has more processing power than a CRAY-1 supercomputer, well, let's just say that a parallel SHARC machine is an awful lot of grunt sitting on your desk. (Before you get too excited, we won't be designing a machine like that. It's far too complex and far too expensive for you to consider and well and truly out of the context of this book!)

The Motorola DSP56000 models are 24-bit processors, primarily intended for audio applications, although they are used in other fields as well. The 24-bit architecture is specifically chosen because 24 bits is a common word size in audio processing. Cochlear uses a DSP56000 in its bionic ear.

Now, although DSPs are beautiful in their intended applications of signal processing, they're also pretty good in general control applications too. An embedded system with a DSP is able to execute sophisticated software and perform advanced algorithms far more efficiently than a conventional processor. Early implementations of embedded DSP systems tended to use the DSP for data processing and include a microcontroller for its ubiquitous functionality. While DSPs are ideal for number crunching, they just weren't particularly good at conventional processor stuff. Having two processors in the one system is not the most efficient design, and so the logical step was a hybrid processor, combining a DSP core with microcontroller functionality.

To this end, the makers of DSPs have developed variants of their DSP architectures specifically intended for embedded applications. They incorporate a DSP core with the type of subsystems normally found in microcontrollers, such as UARTs, SPI, ADCs, and so on. Their instruction sets are also a mixture, incorporating both DSP (data movement and arithmetic) instructions and conventional microprocessor instructions. They are ideal for such applications as motor control (especially in robotics), neural networks and fuzzy control, data compression, digital communications, digital cameras, or any application that is mathematically intensive yet requires small (and relatively cheap) hardware.

In this chapter, we're going to look at the Motorola DSP56800 series of DSP controllers and specifically the DSP56805 processor. We'll see how you design and build a computer based on this chip. DSP56800 processors are specifically designed for implementing advanced digital control and processing in small-scale and low-cost embedded systems. TI and Analog Devices produce comparable processors, and while their architectures may vary, the basic techniques involved in building a computer based upon them are fundamentally the same.

The DSP56800

Unlike the conventional DSP56000 with its 24-bit architecture, the DSP56800s have a 16-bit architecture better suited to small-scale control applications. They are fixed point (integer) only, which is fine for most control applications. If necessary, floating point arithmetic can be synthesized in software.

The architecture is based upon four functional units, each with its own registers, operating independently and in parallel with the other units. These functional units are the *Program Controller*, which is responsible for software execution; the *Address Generation Unit* (*AGU*), which handles bus accesses; the *Data ALU*, which performs the arithmetic operations; and the *Bit-Manipulation Unit* for efficient and rapid bit-based operations.

The independent operation of these units allows for very efficient and fast software execution. While the Data ALU or Bit-Manipulation unit is performing an operation specified by an instruction, the AGU can be generating addresses for the execution of another instruction, while the program controller can be fetching yet another instruction for execution. The instruction set directly supports this parallelism. To accomplish this high internal throughput, the processor has not one, but three internal address buses and four internal data buses (three data buses for the core and one for peripherals). Two operands may be sourced from the internal memory and operated upon in a single instruction. The result is that the architecture achieves a throughput of 40MIPS on an 80MHz clock. That's RISC-like performance with a CISC-like instruction set. In other words, that's a lot of punch.

There's more. It has hardware looping using the `DO` and `REP` instructions. `DO` allows you to specify a block of code (of any size) and have the processor execute it as a loop in hardware. You don't need a counter test and conditional branch instruction at each iteration, saving processor execution overhead. `REP` allows the repetition of a single instruction and `REP`s can be nested inside `DO` loops. As such, you have very versatile looping capability with no overhead. Loops on a DSP are fast!

The programmer's model for the DSP56800 core is shown in Figure 8-1.

The processor has two 36-bit accumulators, a 16x16-bit *Multiply and Accumulate* (*MAC*) unit and a 16-bit *barrel shifter*. The MAC allows you to multiply two numbers and then add the result to a growing total, all with a single instruction. MACs allow for efficient execution of many signal-processing algorithms, as well as neuro-fuzzy code. The barrel shifter allows you to shift up to 16 bits in either direction in a single cycle. So, if you want to shift an operand 15 bits to the left, a conventional processor would require 15 separate shift-left instructions (or one shift-left, a loop, a counter variable, and a conditional test for the loop). The DSP56800, like many DSPs, can perform this operation in just one cycle.

In short, the DSP56800 has very tight and efficient code with high functionality that it executes exceptionally quickly. It is a fast processor around which you can easily design a powerful embedded computer system.

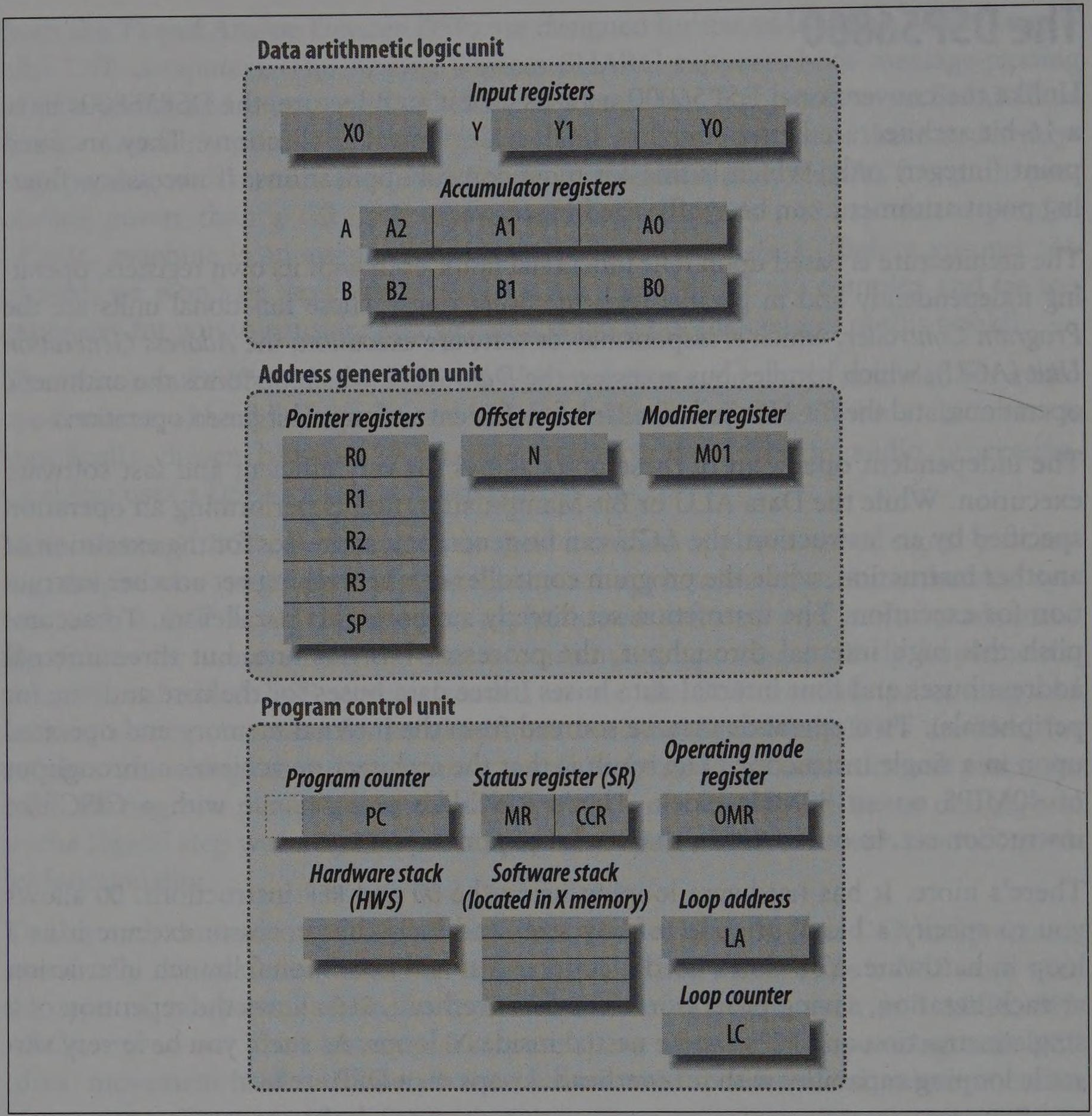


Figure 8-1. DSP56800 programmers' model

We'll look at how you design a system based upon the DSP56805 processor, a member of the DSP56800 family specifically designed for industrial control. The DSP56805 has an internal 1K program RAM, 4K of bootstrap ROM (for loading boot software from an external memory or peripheral), 63K of program flash, 8K of data flash, and 4K of data RAM. The processors also have external data and address buses, so the processor's memory can be expanded well beyond its internal resources. It has a 64K x 16-bit address space, giving access to 128K (bytes) of external memory. (Some DSP568xx processors have significantly larger address spaces than this.) The DSP5685x series can support up to 2M of program memory and up to 8M of data memory. The DSP56800 processors also provide the ability to separate data and program spaces, thereby doubling the external address space. The processor also has a programmable wait-state generator, simplifying interfacing to external devices. The

generator may be programmed to provide 0, 4, 8, or 12 wait states for accesses to a given device.

DSP56800s in general come with a range of built-in peripherals, including SPI ports (sometimes two), several 16-bit general-purpose timers, a watchdog timer (called a *Computer Operating Properly*, or *COP*, timer by Motorola), a timer for real-time operation, a *Synchronous Serial Interface* (*SSI*) for accessing audio codecs (combined ADCs and DACs) and other DSPs, and general-purpose I/O lines. The DSP56805 adds two six-channel *Pulse Width Modulation* (*PWM*) units (Chapter 12) for motor control and other uses, two four-channel ADCs at a resolution of 12 bits per channel, and two quadrature decoders for measuring motor positions (covered in Chapter 12). It also has a CAN networking module (discussed in Chapter 11), two serial ports (called Serial Communication Interfaces, or SCIs, by Motorola), and 14 dedicated and 18 shared I/O lines.

The processors operate from a supply voltage of between 3.0V and 3.6V but have 5V-tolerant inputs, making interfacing to a wide variety of devices easy. (Other DSP56800s may operate on a supply voltage of between 4.57V and 5.5V, depending on the particular chip.) The processor has several low-power and sleep modes, making it ideal for battery-powered systems.

All DSP56800 processors incorporate a *JTAG* (*Joint Test Action Group*) port for interfacing to specialized debugging instruments. The JTAG port also allows direct access to the processor's onboard flash program memory, making the job of downloading new code simple and fast.

> The JTAG port allows for real-time debugging of hardware and software. It allows you to single-step or multi-step through code running directly on the target system. You can individually (and manually) toggle signal lines of the processor to test external subsystems in the computer (also known as *boundary scan*). You can set breakpoints both at locations in code or for when a particular address (or device) is accessed. The JTAG port allows you to examine and modify registers and memory locations. To utilize the JTAG interface, you need to have support tools that are JTAG compliant. For more information, refer to *IEEE standard 1149.1a*.

A block diagram of the DSP56805 is shown in Figure 8-2.

All in all, quite a nice processor. So, let's look at how you build a system based upon one. For simplicity, I'll look at each subsystem in turn.

A DSP56805-Based Computer

The DSP56805 has nine power pins. Each of these must be decoupled to ground using 100nF ceramic capacitors. Each capacitor should be placed as close as possible to its respective power pin. Since this processor can operate at a relatively high speed, and can therefore generate a lot of noise, a four-layer circuit board is preferred

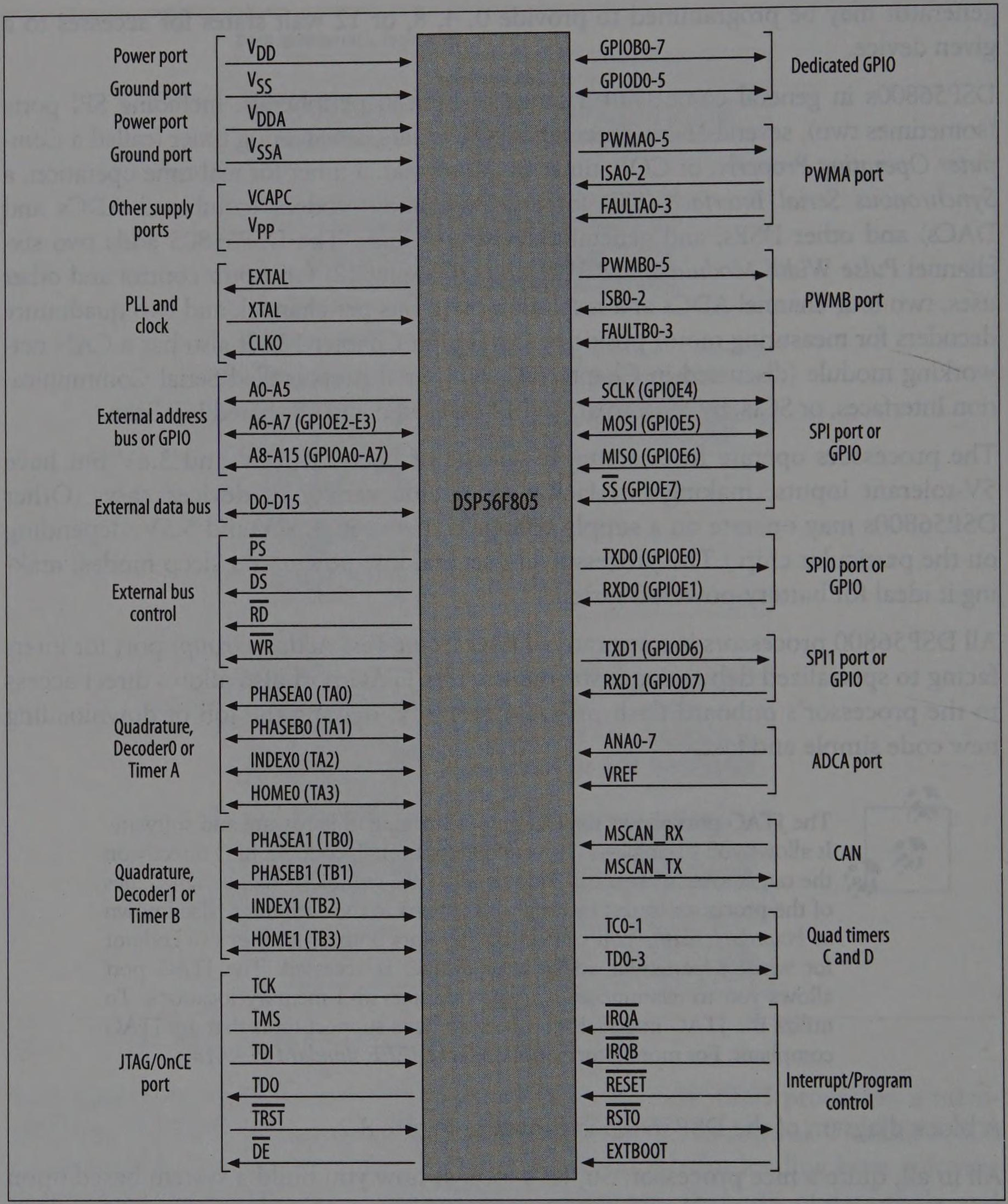


Figure 8-2. DSP56805 block diagram

for construction. (See Chapter 4 for more information.) As with any design, any unused inputs must be tied inactive.

Oscillator

Like all processors, the DSP56805 requires a clock signal. The processor can operate from an oscillator frequency of up to 80MHz (giving 40MIPS) or as slow as a few

MHz to save power. The processor may even have its clock completely stopped (so-called DC operation, meaning that the clock is no longer an AC signal) to further save power. (This processor's sibling, the DSP56801, has a complete internal oscillator and so requires no external clock generation circuit.)

The processor has a built-in oscillator circuit, requiring only an external crystal in the range of 4MHz to 8MHz and support components (Figure 8-3). From this low crystal frequency, the processor internally synthesizes a clock speed of between 40MHz and 110MHz under software control. Note that while the clock generation circuit is able to produce 110MHz, the processor isn't able to operate at the speed. So keep the speed below 80MHz, and the processor, your software, and you will all be happy.

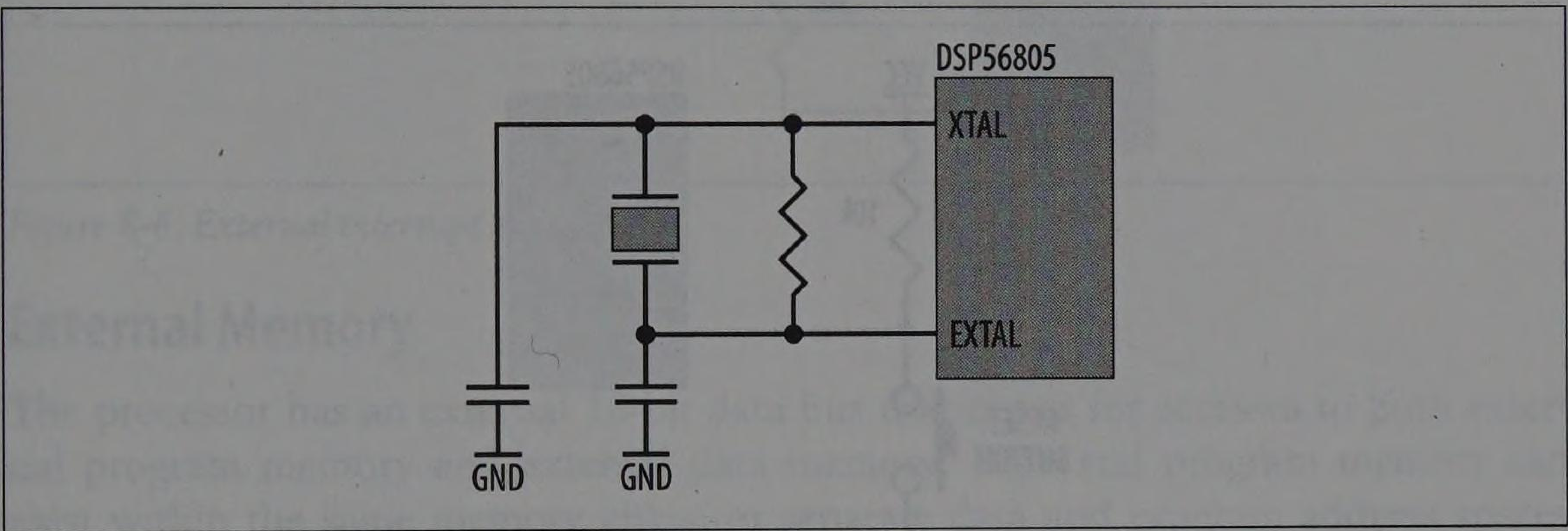


Figure 8-3. Crystal oscillator circuit

In a typical application, the crystal frequency is 8MHz, with a resistor value of 10MΩ. Decoupling capacitors are approximately 15pF or so. However, the values of the resistor and capacitors required can vary, so make sure you check the technical data from the crystal manufacturer. It will tell you specifically what values to use for a particular crystal.

Alternatively, you could use an external oscillator module to generate the processor's clock (Figure 8-4). The module's output is connected to the **XTAL** input of the processor. When operating in this configuration, **EXTAL** must be connected to ground.

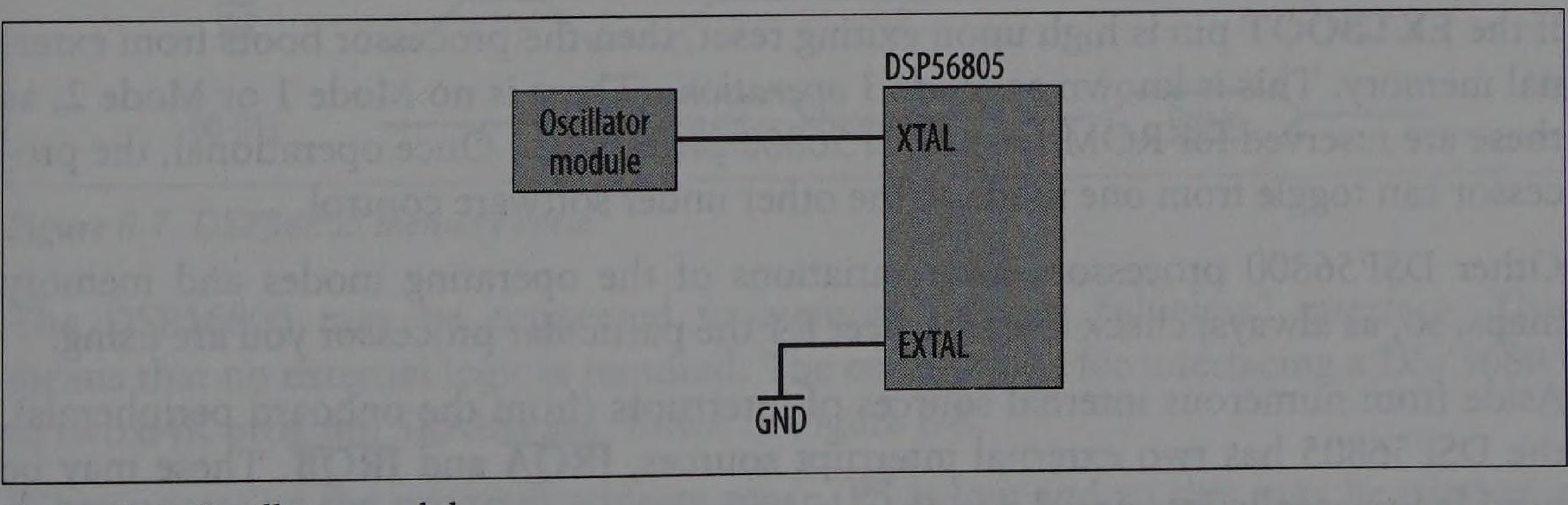


Figure 8-4. Oscillator module

Reset and Interrupts

The DSP56805 has an internal power-on circuit to correctly start up the processor. It also has a watchdog reset circuit, driven by an internal timer, to recover the processor from a software crash. So, all we need to do is to provide our system with an external reset so that we can manually restart the machine by pressing a button. Normally, such a reset circuit would need to debounce the button press and also ensure that the reset state was held for a minimum period of time. On the DSP56805, life is much simpler. The processor incorporates internal debounce circuitry on its **RESET** input. Further, it has circuitry that ensures that a reset is held for the appropriate duration. So, our external reset circuit is simply a push button and a pull-up resistor (Figure 8-5). What could be simpler?

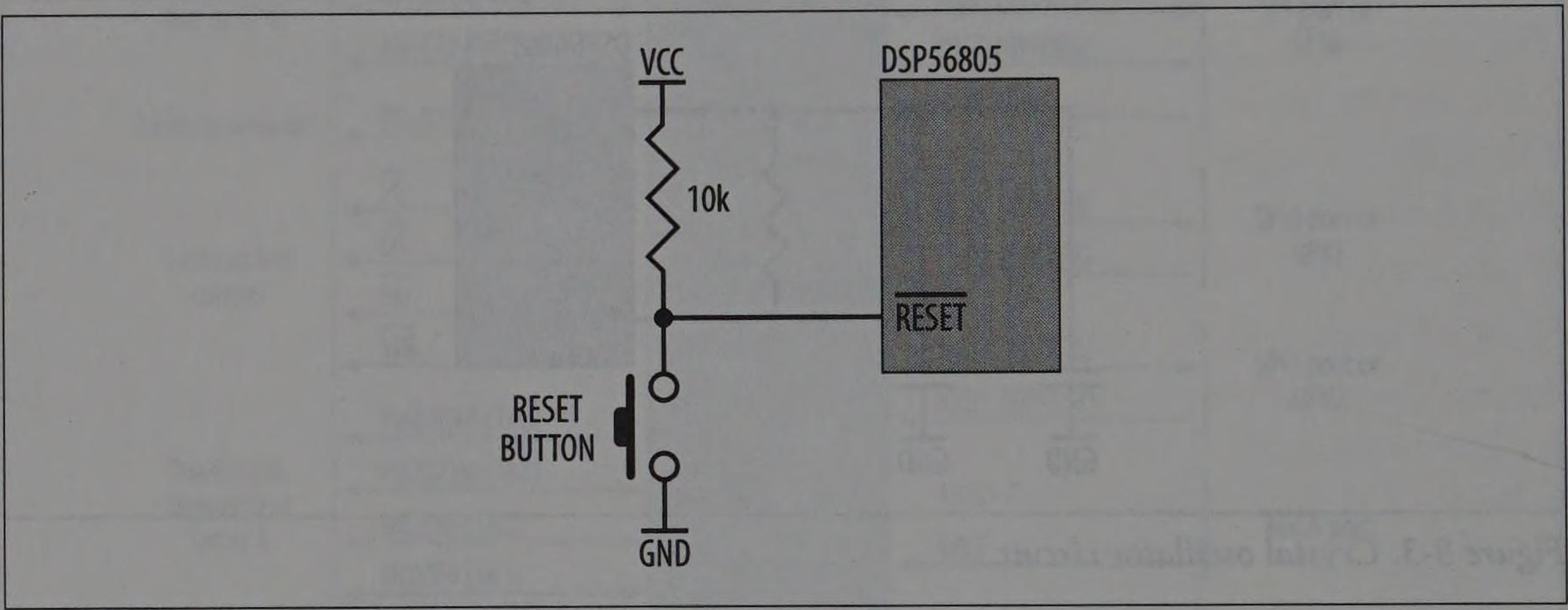


Figure 8-5. External reset on a DSP56805

The DSP56805 can boot from external memory or from its internal ROM for single-chip operation. An input pin, **EXTBOOT**, is sampled as the processor comes out of reset. If **EXTBOOT** is pulled low, the processor executes code from the internal ROM. This is known as *Mode 0 operation*. There are two forms of Mode 0. Mode 0A maps all memory as internal, whereas Mode 0B maps the lower 32K words (64K bytes) of the address space as internal and the upper 32K words as external. Mode 0A is the default mode and Mode 0B may be entered only under software control.

If the **EXTBOOT** pin is high upon exiting reset, then the processor boots from external memory. This is known as *Mode 3 operation*. (There is no Mode 1 or Mode 2, as these are reserved for ROM-based DSP56800 processors.) Once operational, the processor can toggle from one mode to the other under software control.

Other DSP56800 processors have variations of the operating modes and memory maps, so, as always, check the datasheet for the particular processor you are using.

Aside from numerous internal sources of interrupts (from the onboard peripherals), the DSP56805 has two external interrupt sources, **IRQA** and **IRQB**. These may be used by externally interfaced peripherals (or even external systems) to gain the processor's attention. Whether they are connected to an external interrupt source or

not, they require an external pull-up resistor. In the example given (Figure 8-6), $\overline{\text{IRQA}}$ has an interrupt source from a peripheral, while $\overline{\text{IRQB}}$ is unused.

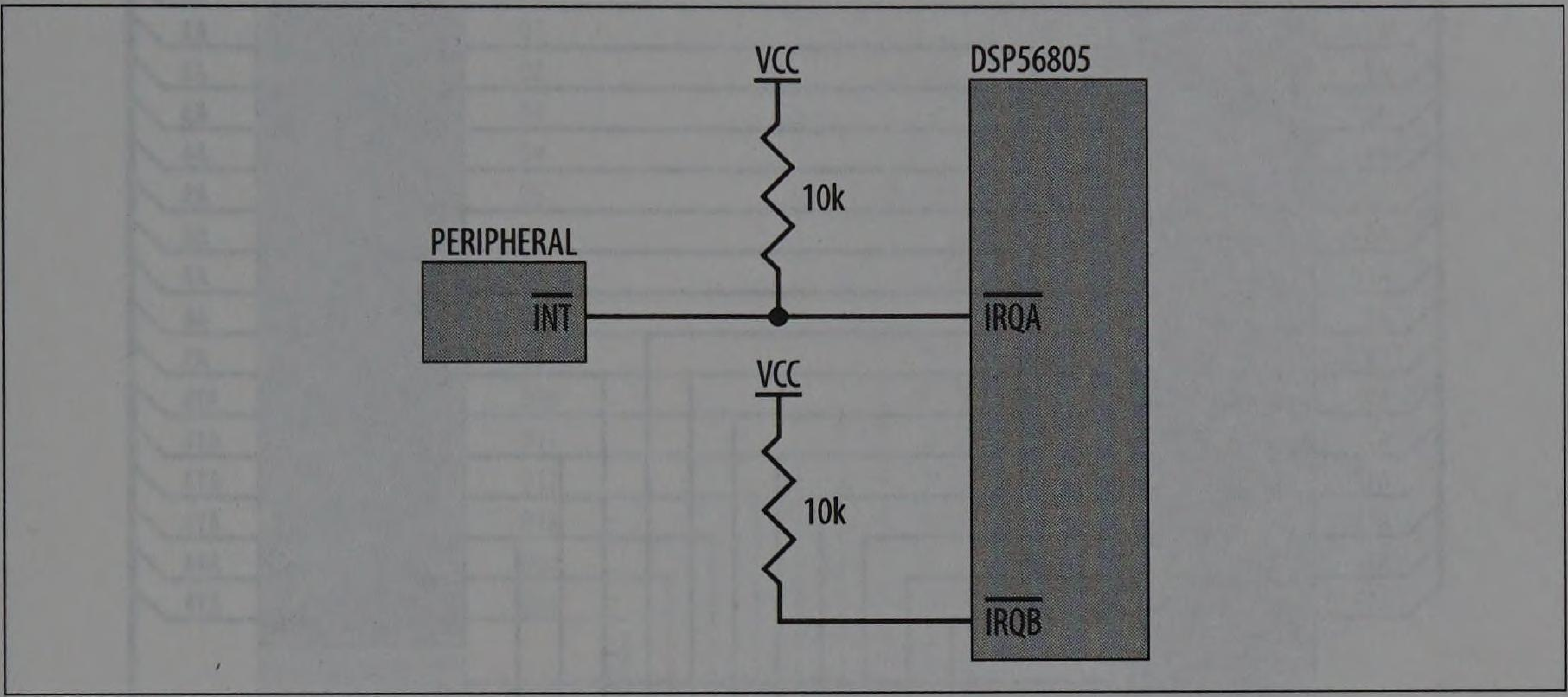


Figure 8-6. External interrupt sources

External Memory

The processor has an external 16-bit data bus that serves for accesses to both external program memory and external data memory. Data and program memory can exist within the same memory chips, or separate data and program address spaces may be implemented. The processor has two outputs, $\overline{\text{PS}}$ (Program Strobe) and $\overline{\text{DS}}$ (Data Strobe), which indicate the type of memory access.

The timing for a DSP56805 write cycle followed by a read cycle is shown in Figure 8-7. Since the processor has a programmable wait-state generator, external memory devices or peripherals of varying response times may be accommodated.

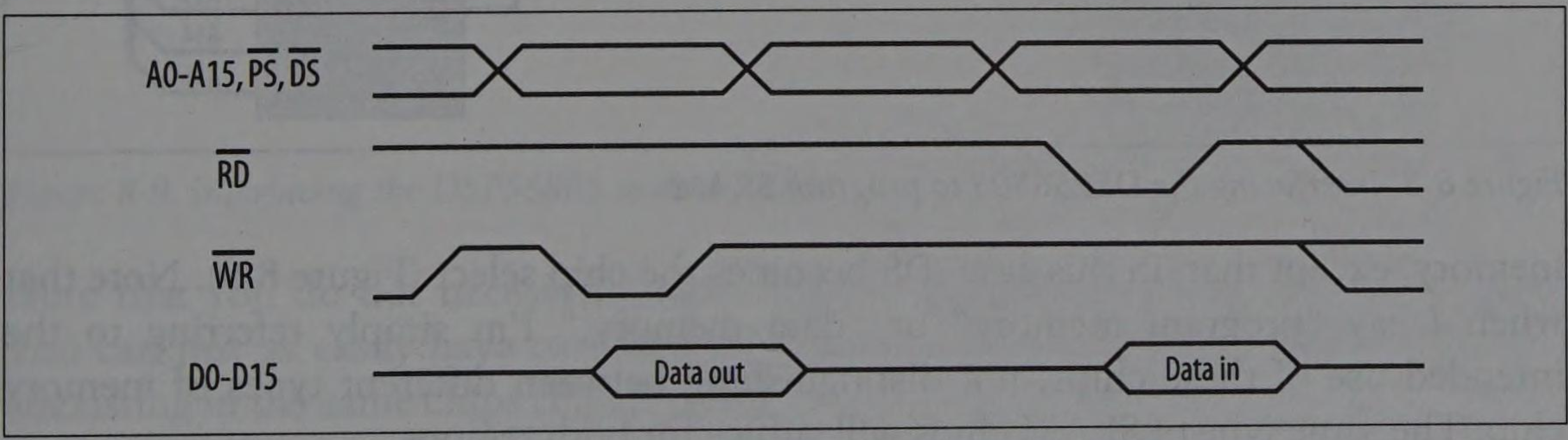


Figure 8-7. DSP56805 memory cycle

The DSP56805 may be connected to memory using a "glueless" interface. This means that no external logic is required. The connections for interfacing a DSP56805 to two 64K program SRAMs are shown in Figure 8-8.

When accessing the program address space, $\overline{\text{PS}}$ is low and so this may be used as a chip select to the SRAMs. Similarly, the same configuration may be used for data

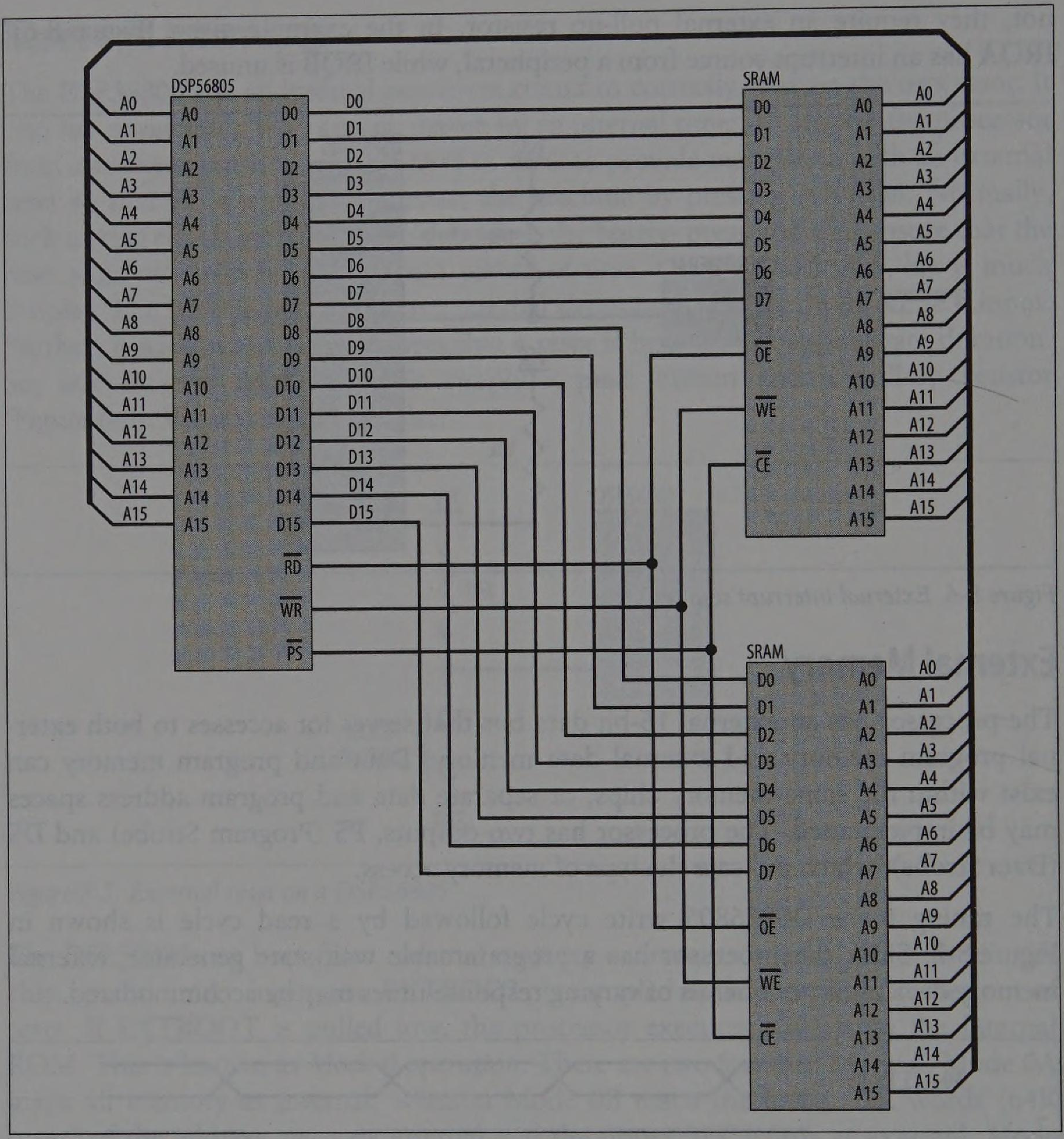


Figure 8-8. Interfacing the DSP56805 to program SRAM

memory, except that, in this case, $\overline{\text{DS}}$ becomes the chip select (Figure 8-9). Note that when I say "program memory" or "data memory," I'm simply referring to the intended use of these chips, not distinguishing between different types of memory chip. The same type of SRAM chips will suffice for both regions.

So, our DSP56805 computer has four SRAM chips in total, evenly divided between program memory and data memory. Each region has 64K x 16 bytes (two 8-bit memory chips), giving a total of 128K bytes of program space and 128K bytes of data memory. The total memory for our system is therefore 256K bytes. If more data memory is required, memory banking may be used to increase the available space, as we saw in Chapter 6 with the AT90S8515.

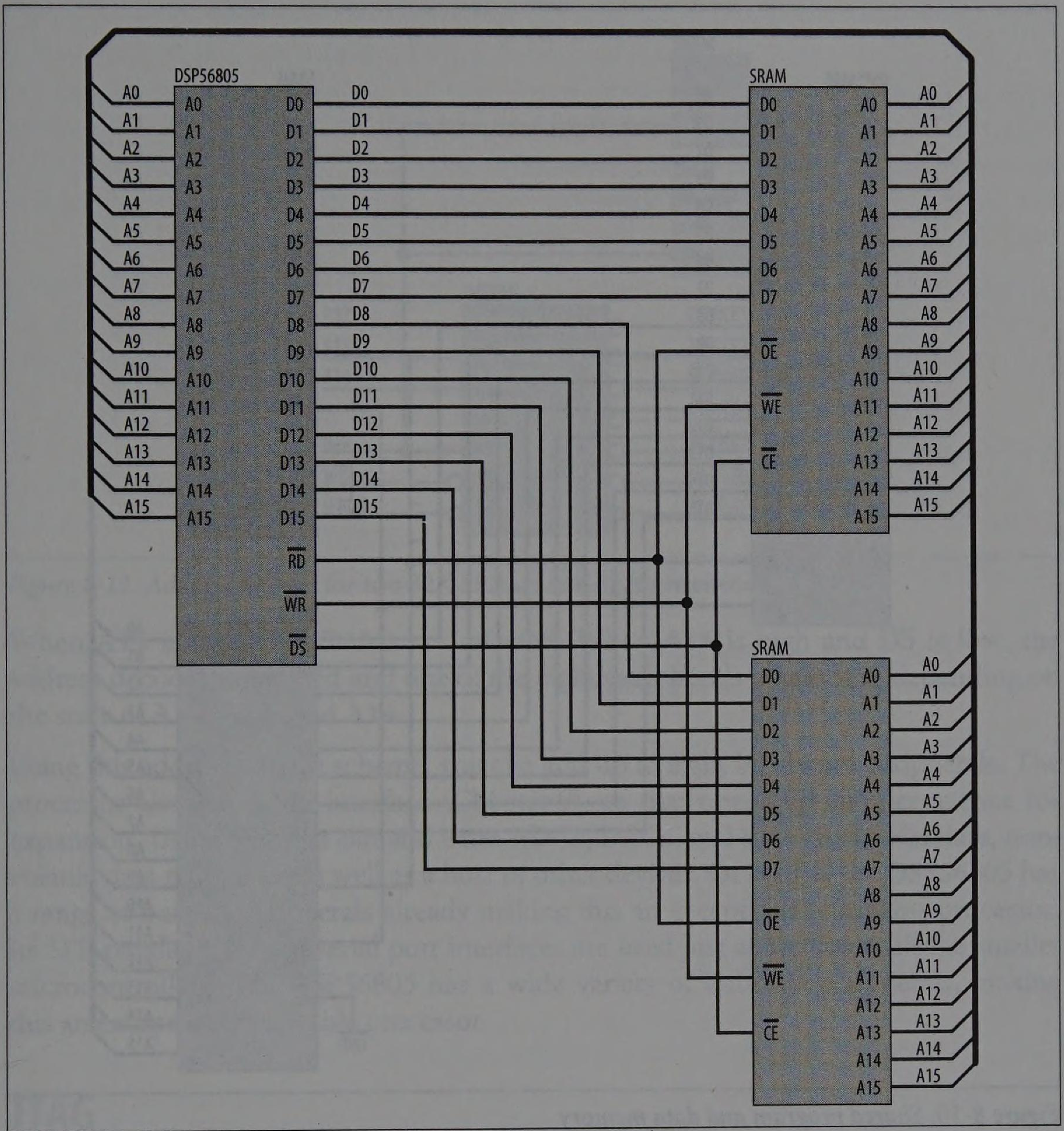


Figure 8-9. Interfacing the DSP56805 to data SRAM

Note that you do not necessarily have to have separate program and data spaces. You can just as easily have two SRAMs in total, with the program and data spaces coexisting in the same chips (Figure 8-10).

In this case, both $\overline{\textbf{PS}}$ and $\overline{\textbf{DS}}$ are ignored, since we are no longer distinguishing between data and program spaces. The chip enable ($\overline{\textbf{CE}}$) inputs of the SRAMs are simply tied to ground, so that these devices are permanently enabled. This will work since an SRAM will respond only if $\overline{\textbf{CE}}$ is low *and* either the output enable ($\overline{\textbf{OE}}$) or the write enable ($\overline{\textbf{WE}}$) go low as well. So, in this example, it is the output enable or write enable that will activate the SRAMs. Note that permanently enabling an SRAM will increase its power

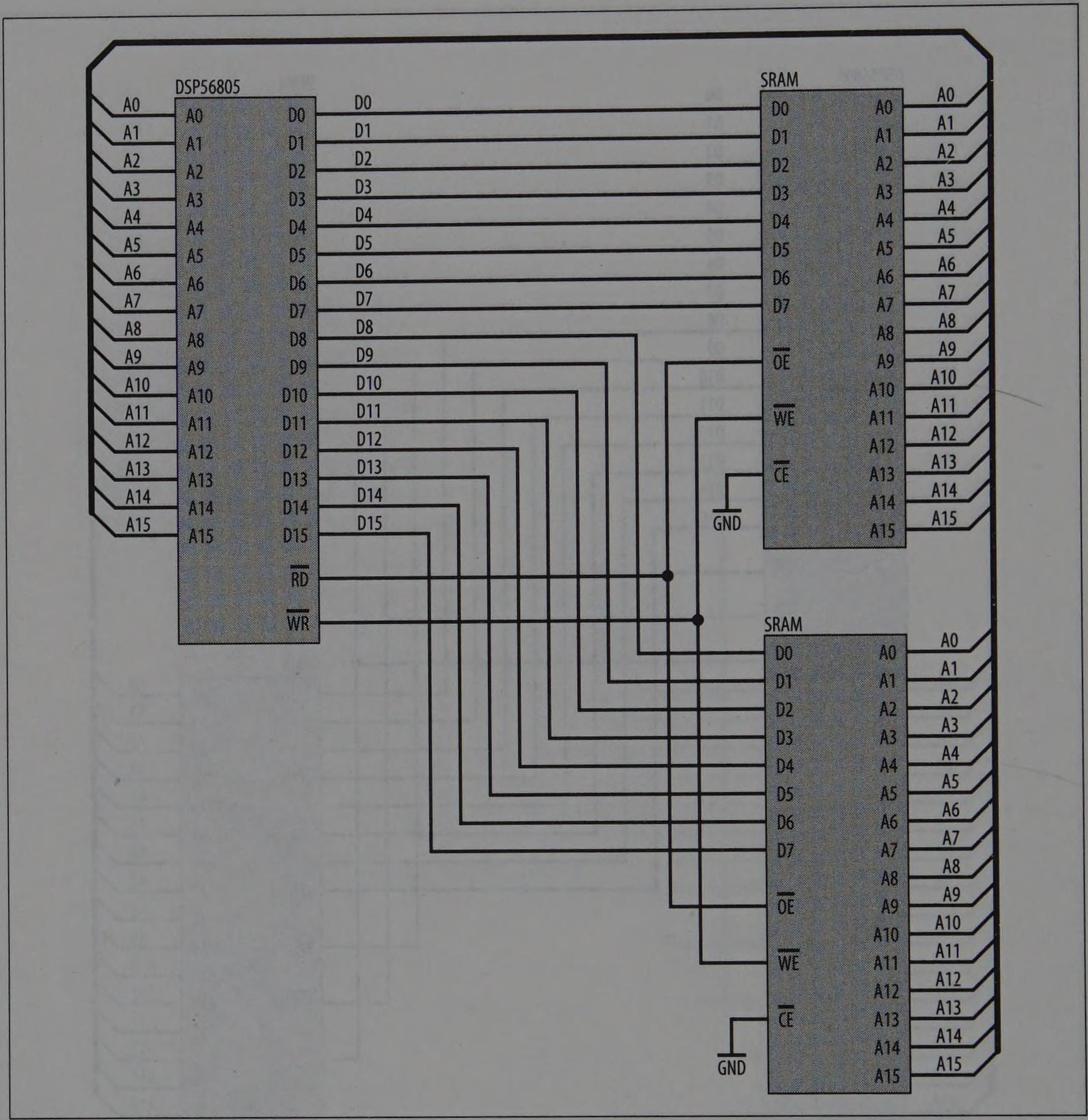


Figure 8-10. Shared program and data memory

consumption. Of course, we could just as easily combine $\overline{\text{DS}}$ and $\overline{\text{PS}}$ so that either going low will enable the SRAMs, but this requires extra logic and it really isn't necessary.

If you have different types of devices within your memory space, such as a smaller data SRAM and some peripherals, then you must include $\overline{\text{DS}}$ as part of the chip enable for the SRAMs and peripherals. The most logical way to do this is to use $\overline{\text{DS}}$ as the enable to your address decoder, which in turns selects the appropriate device. Note that it must be $\overline{\text{DS}}$ for accessing peripherals, since you can't execute code directly out of a peripheral!

An example address decoder is shown in Figure 8-11. This will select either two 32K SRAMs or one of eight peripherals within the data space.

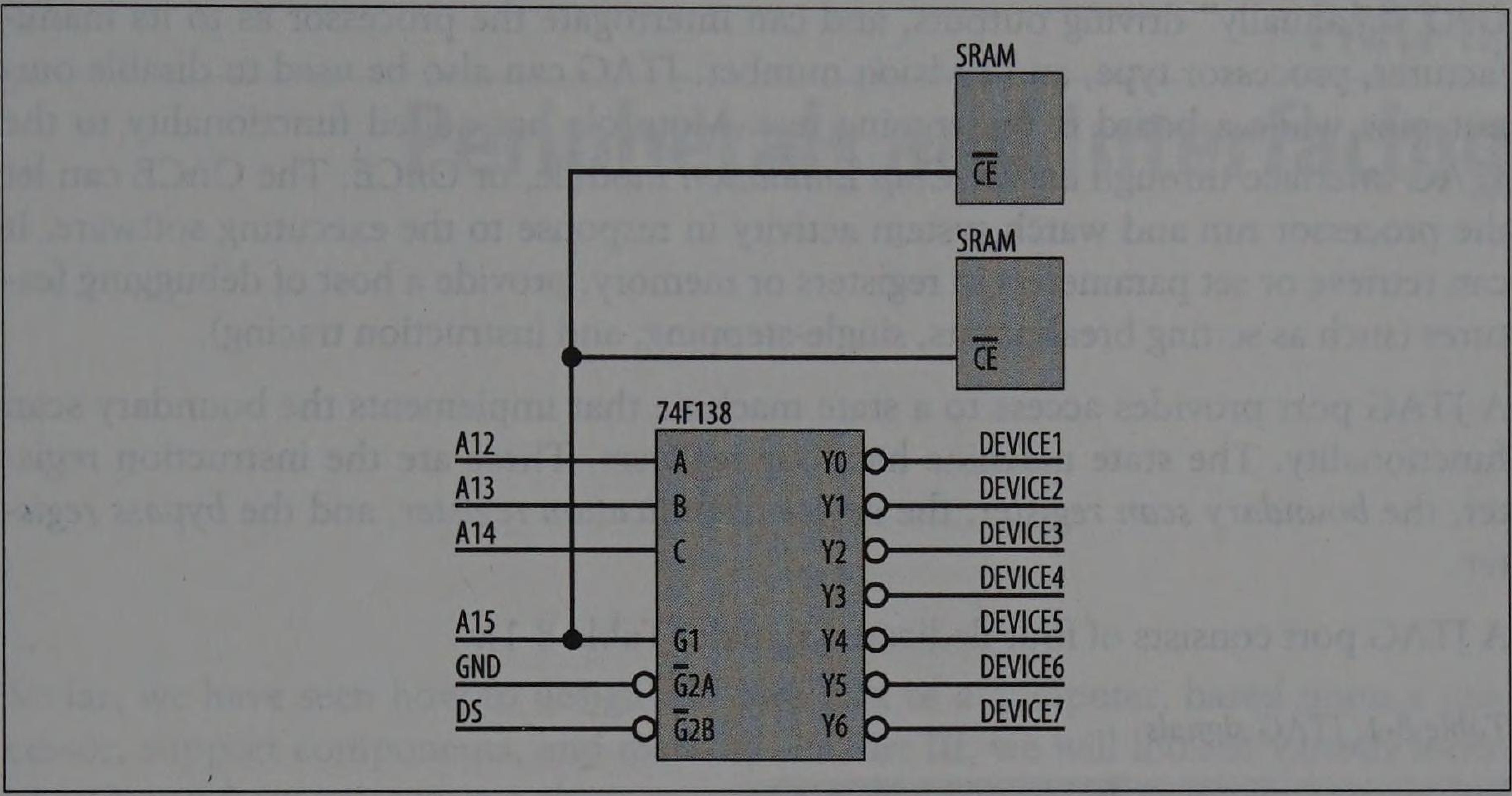


Figure 8-11. Address decoder for two 32K SRAMs and eight peripherals

When **A15** is low, the SRAMs are selected. When **A15** is high and $\overline{\textbf{DS}}$ is low, the address decoder is enabled and one of the eight peripherals is selected, depending on the state of **A12**, **A13**, and **A14**.

Using this address decode scheme, you can add up to eight bus-based peripherals. The processor also has a SPI interface (Chapter 9), so that opens up another avenue for expansion. Using SPI, you can add extra ADCs, DACs, real-time clock calendars, non-volatile data memories, as well as a host of other devices. Of course, the DSP56805 has a range of built-in peripherals already making this an exceptionally capable processor. Its SPI, parallel I/O, and serial port interfaces are used just as we saw with the smaller microcontrollers. The DSP56805 has a wide variety of onboard peripherals, making this an exceptionally capable processor.

JTAG

As mentioned earlier, the JTAG port (sometimes also known as a *Test Access Port*, or *TAP*) provides access to the internals of the processor and, through it, the rest of the computer system. JTAG is defined under IEEE standard *1149.1a-1993 Standard Test Access Port and Boundary Scan Architecture*. Commercially available test suites use JTAG to provide in-circuit debug capability. The adventurous among you can also drive JTAG "manually," using the information in that standard document.

JTAG uses a technique known as *boundary scan* to probe the circuit connections between peripherals and memories and the microprocessor. It does this by asserting outputs (independently of the CPU) and reading the response from external devices on input pins. It is useful for testing not only interconnections on the PCB but also design verification and even correct timing. JTAG can operate independently of the

CPU, "manually" driving outputs, and can interrogate the processor as to its manufacturer, processor type, and revision number. JTAG can also be used to disable output pins while a board is undergoing test. Motorola has added functionality to the JTAG interface through an *On-Chip Emulation* module, or *OnCE*. The OnCE can let the processor run and watch system activity in response to the executing software. It can retrieve or set parameters in registers or memory, provide a host of debugging features (such as setting breakpoints, single-stepping, and instruction tracing).

A JTAG port provides access to a state machine that implements the boundary scan functionality. The state machine has four registers. These are the instruction register, the *boundary scan register*, the *device identification register*, and the *bypass register*.

A JTAG port consists of four dedicated signals (Table 8-1).

Table 8-1. JTAG signals

Signal name	Function
TDI	Test data input
TDO	Test data output
TMS	Test mode select
TCK	Test clock

If you think those signals sound suspiciously like a synchronous serial interface, you'd be right, for that is exactly what JTAG is.

Motorola adds additional signals to the standard JTAG set. Specifically, **TRST** (Test Reset) to reset the JTAG state machine and **DE** (Debug Event), which is equivalent to an interrupt output, indicating that an event (such as a breakpoint) has happened in the OnCE module.

JTAG is principally intended for debugging purposes, but since it gives you complete control of the processor's internals, it can also be used for reprogramming the internal program flash. The Motorola application note (AN1935/D) *Programming On-Chip Flash Memories of DSP56F80x DSPs Using the JTAG/OnCE Interface*, available from the Motorola web site, contains full details on the process involved, as well as sample source code and examples.

The Motorola Software Development Kit, based on the CodeWarrior C compiler, for the DSP56800 series provides both software and hardware tools for programming these processors.

So far, we've looked at various processors and how we design computers based upon them. We have yet to look at how you interface them to the real world and do something useful. In Part III of this book, we'll take a tour of peripherals and see how to give purpose to our embedded machines.

PART III

Peripherals and Interfacing

So far, we have seen how to design the core part of a computer, based upon a processor, support components, and memory. In Part III, we will look at various forms of I/O and how we can use them to connect our embedded computer to the real world.

In Chapter 9, we will see how to add additional peripherals using two simple interfaces found in many embedded processors.

Chapter 10 shows us how to connect our embedded system to other computers using serial interfaces. We'll see how to implement an RS-232C serial port and learn how we can use this to interface our embedded machines to PCs, terminals, and modems. We'll also look at RS-422, a more robust type of serial interface, and finally we'll see how to communicate with light using IrDA.

Chapter 11 extends these concepts, and we learn how to add network interfaces to our designs. We will look at three networks: RS-485, CAN, and Ethernet.

Chapter 12 covers interfacing to the analog world. We look at the basic principles of sampling an analog signal, and then we'll see how we can amplify a very small analog signal prior to sampling. We'll take a look at analog-to-digital conversion and then how to interface some simple sensors to our embedded system. We'll see how to measure temperature, light, pressure, vibration, and magnetic fields. We'll also see how to use a technique known as quadrature encoding to measure the angular position of a rotating object. Finally, we'll look at how we can convert a digital value back into an analog signal.

CHAPTER 9

Adding Peripherals Using SPI and I²C

Thirty spokes meet at a nave;
Because of the hole we may use the wheel.
Clay is molded into a vessel;
Because of the hollow we may use the cup.
Walls are built around a hearth;
Because of the doors we may use the house.
Thus tools come from what exists,
But use from what does not.

—Lao Tse
Tao Te Ching

In this chapter, we'll look at two simple interfaces used to connect peripheral chips to microcontrollers, within a single embedded system. These interfaces allow you to connect devices such as real-time clocks, nonvolatile memories for parameter storage, sensor interfaces, and much more. The interfaces are easy to use and cheap to implement, making them ideal for small embedded applications. Some microcontrollers incorporate both types of interface, whereas others may have only one or the other. Which to use really depends on what your processor has to offer and the requirements of the particular peripheral you're using.

Serial Peripheral Interface

The *Serial Peripheral Interface* (known as *SPI*) was developed by Motorola to provide a low-cost and simple interface between microcontrollers and peripheral chips. (SPI is sometimes also known as a *four-wire interface*.) It can be used to interface to memory (for data storage), analog-digital converters, digital-analog converters, real-time clock calendars, LCD drivers, sensors, audio chips, and even other processors. The range of components that support SPI is large and growing all the time.

Unlike a standard serial port (which is covered in Chapter 10), SPI is a synchronous protocol in which all transmissions are referenced to a common clock, generated by

the *master* (processor). The receiving peripheral (*slave*) uses the clock to synchronize its acquisition of the serial bit stream. Many chips may be connected to the same SPI interface of a master. A master selects a slave to receive by asserting the slave's chip select input. A peripheral that is not selected will not take part in a SPI transfer.

SPI uses four main signals: Master Out Slave In (**MOSI**), Master In Slave Out (**MISO**), Serial CLocK (**SCLK** or **SCK**), and Chip Select ($\overline{\textbf{CS}}$) for the peripheral. Some processors have a dedicated chip select for SPI interfacing called Slave Select ($\overline{\textbf{SS}}$).

MOSI is generated by the master and is received by the slave. On some chips, **MOSI** is labeled simply as Serial In (**SI**) or Serial Data In (**SDI**). **MISO** is produced by the slave, but its generation is controlled by the master. **MISO** is sometimes known as Serial Out (**SO**) or Serial Data Out (**SDO**) on some chips. The chip select to the peripheral is normally generated by simply using a spare I/O pin of the master. Figure 9-1 shows a microprocessor interfaced to a peripheral using SPI.

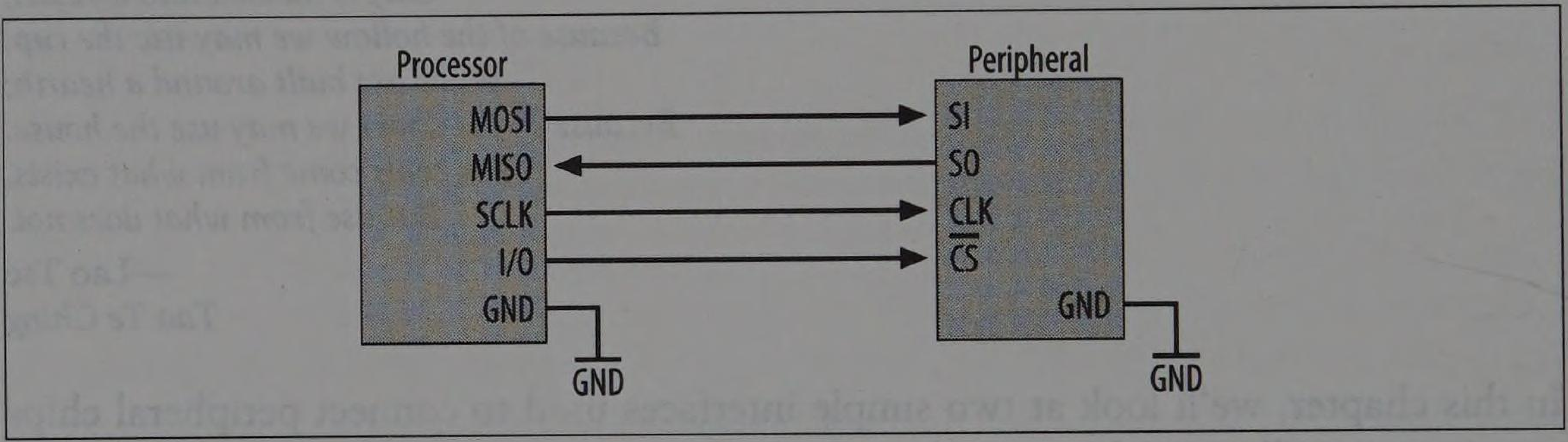


Figure 9-1. Basic SPI interface

Both masters and slaves contain a serial shift register. The master starts a transfer of a byte by writing it to its SPI shift register. As the register transmits the byte to the slave on the **MOSI** signal line, the slave transfers the contents of *its* shift register back to the master on the **MISO** signal line (Figure 9-2). In this way, the contents of the two shift registers are exchanged. Both a write and a read operation are performed with the slave simultaneously. SPI can therefore be a very efficient protocol.

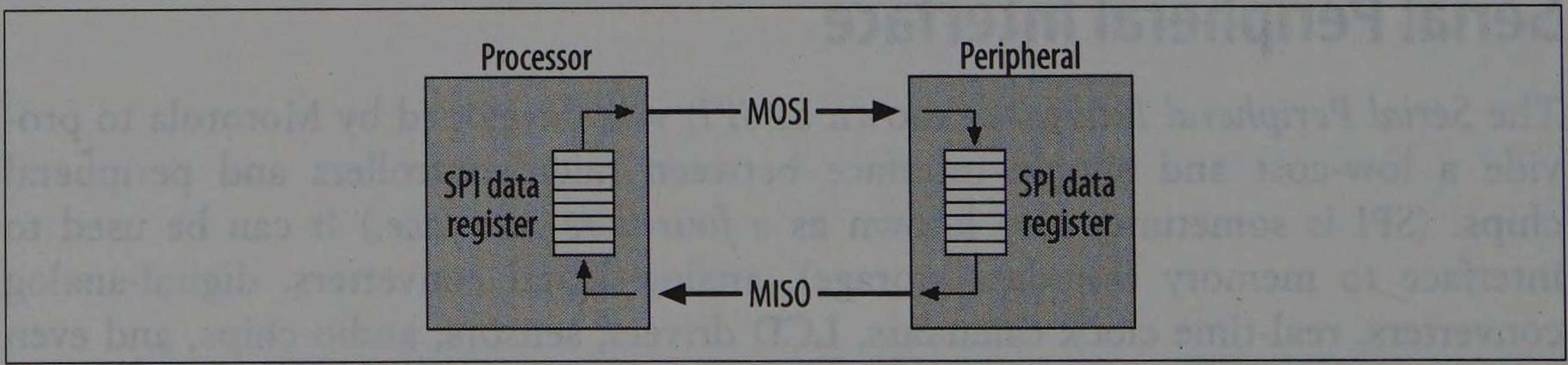


Figure 9-2. SPI transmission

If only a write operation is desired, the master just ignores the byte it receives. Conversely, if the master just wishes to read a byte from the slave, it must transfer a dummy byte in order to initiate a slave transmission.

Some peripherals can handle multiple byte transfers, with a continuous stream of data shifted from the master. Many memory chips with SPI interfaces work this way. With this type of transfer, the chip select for the SPI slave must remain low for the entire duration of the transmission. For example, a memory chip might expect a "write" command to be followed by four address bytes (starting address), then the data bytes to be stored. A single transfer may involve the shifting of a kilobyte or more of information.

Other slaves need only a single byte (for example, a command byte for an analog-digital converter), and some even support being daisy-chained together (Figure 9-3).

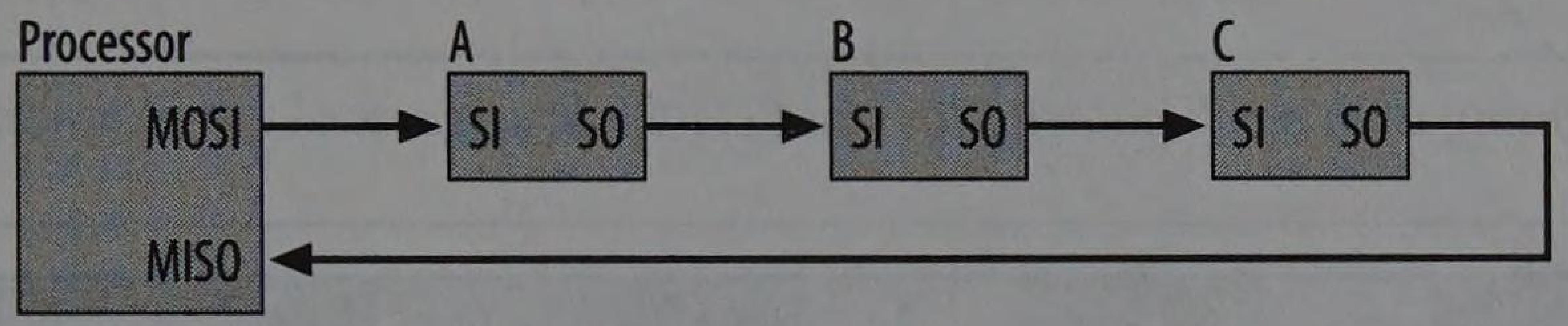


Figure 9-3. Daisy-chaining three SPI devices

In this example, the master processor transmits 3 bytes out of its SPI interface. The first byte is shifted into slave A. As the second byte is transferred to slave A, the first byte is shifted out of slave A and into slave B. Similarly, as the third byte is shifted into slave A, the second byte is shifted into slave B, and the first byte is shifted into slave C. If the master wishes to read a result from slave A, it must again transfer a 3-byte (dummy) sequence. This will move the byte from slave A into slave B, then into slave C, and finally into the master. In the process, the master also receives bytes from slave C and slave B in turn.

Note that daisy-chaining won't necessarily work with all SPI devices, especially ones that require multibyte transfers (such as memory chips). Again, it's a case of checking the slave chips' datasheets carefully to determine what you can and can't do. If the datasheet doesn't explicitly mention daisy-chaining, then it's a fair bet that the device doesn't support it.

SPI has four modes of operation, depending on clock polarity and clock phase. For low clock polarity, the clock (**SCK**) is low when idle and toggles high during a transfer. When configured for high clock polarity, the clock is high when idle and toggles low during a transfer.

The two clock phases are known as *clock phase zero* and *clock phase one*. For clock phase zero, **MOSI** and **MISO** outputs are valid on the rising edge of the clock (**SCK**) if the clock polarity is low (Figure 9-4). If the clock polarity is high, these outputs are valid on the falling edge of **SCK**, for clock phase zero (Figure 9-5). The X bit output on **MISO** is an undefined extra bit and is a consequence of the SPI interface. You don't need to worry about it as the SPI interfaces ignore it.

Conversely, for clock phase one, the opposite is true. **MOSI** and **MISO** are valid on the falling edge of the clock if clock polarity is low (Figure 9-6). They are valid on the rising edge of the clock if the clock polarity is high (Figure 9-7).

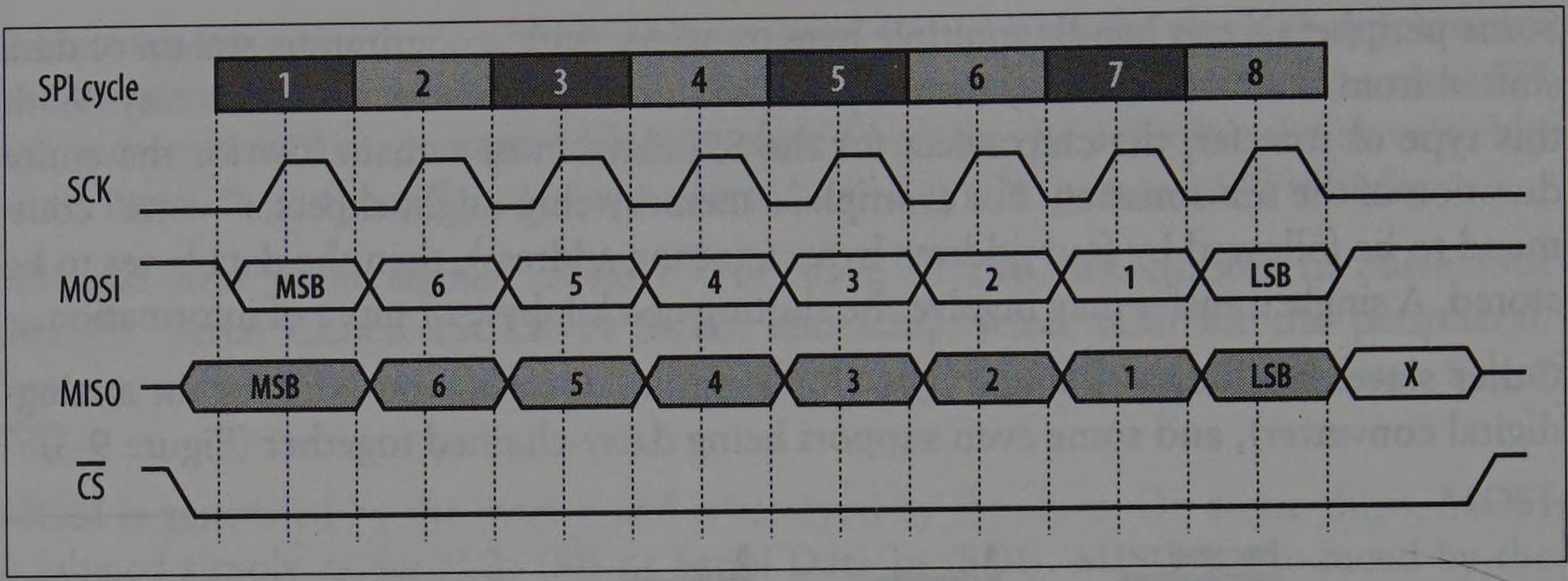


Figure 9-4. SPI timing with clock polarity low and clock phase zero

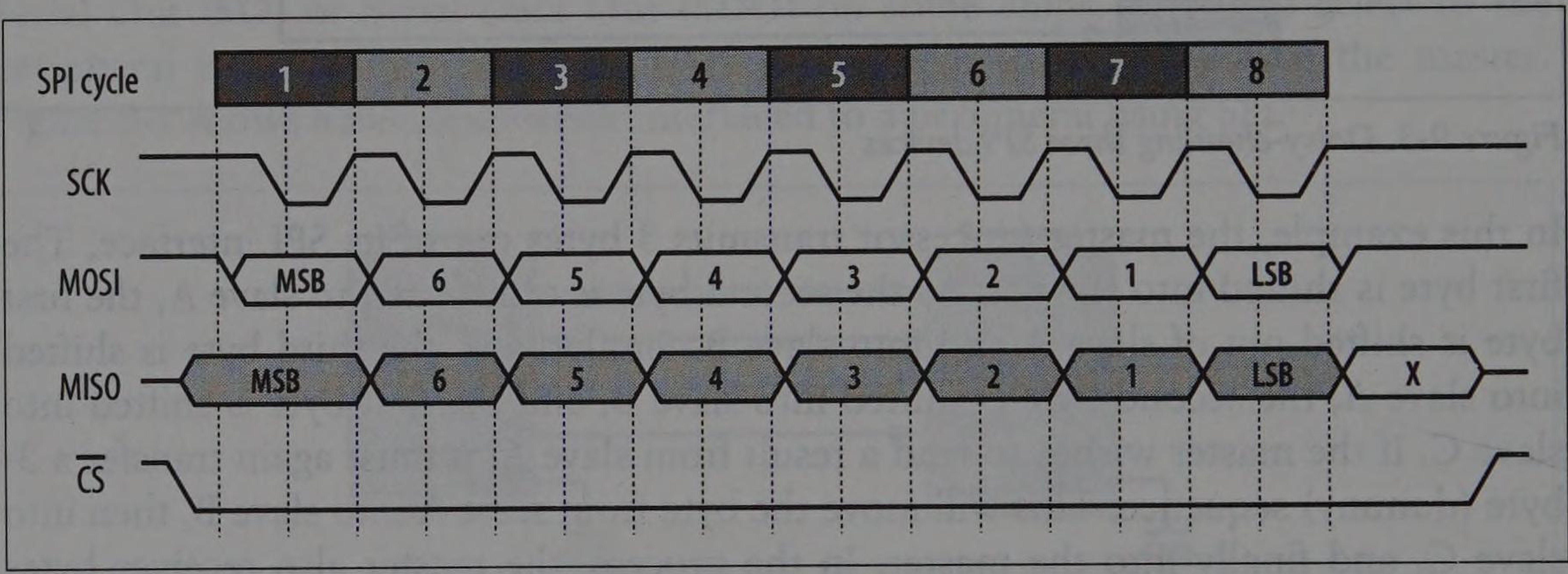


Figure 9-5. SPI timing with clock polarity high and clock phase zero

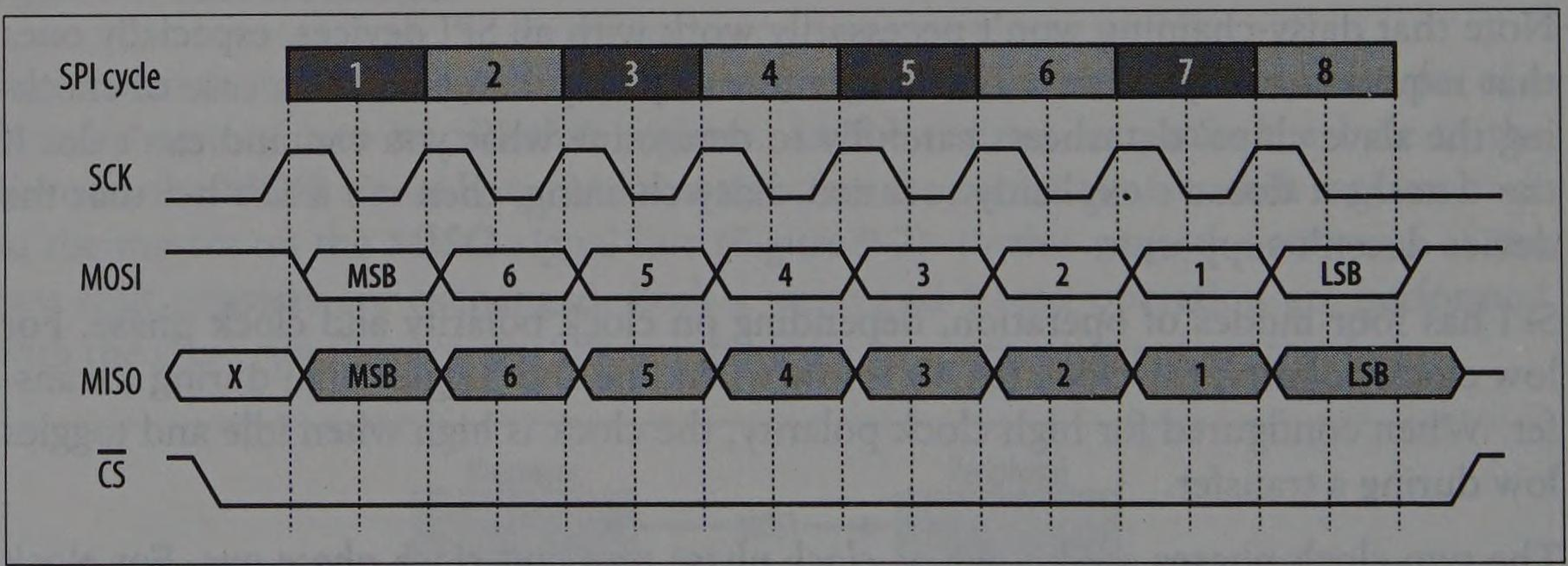


Figure 9-6. SPI timing with clock polarity low and clock phase one

SPI-based Clock/Calendar

There is a wide variety of SPI devices available, and we'll be looking at several in the coming chapters. In the meantime, to see how a SPI interface is used to add a peripheral to a microcontroller, let's look at interfacing a processor to a clock-calendar chip. Such chips contain an oscillator module driven by a crystal, just like a processor. The oscillator module ticks over internal counters that track milliseconds, seconds,

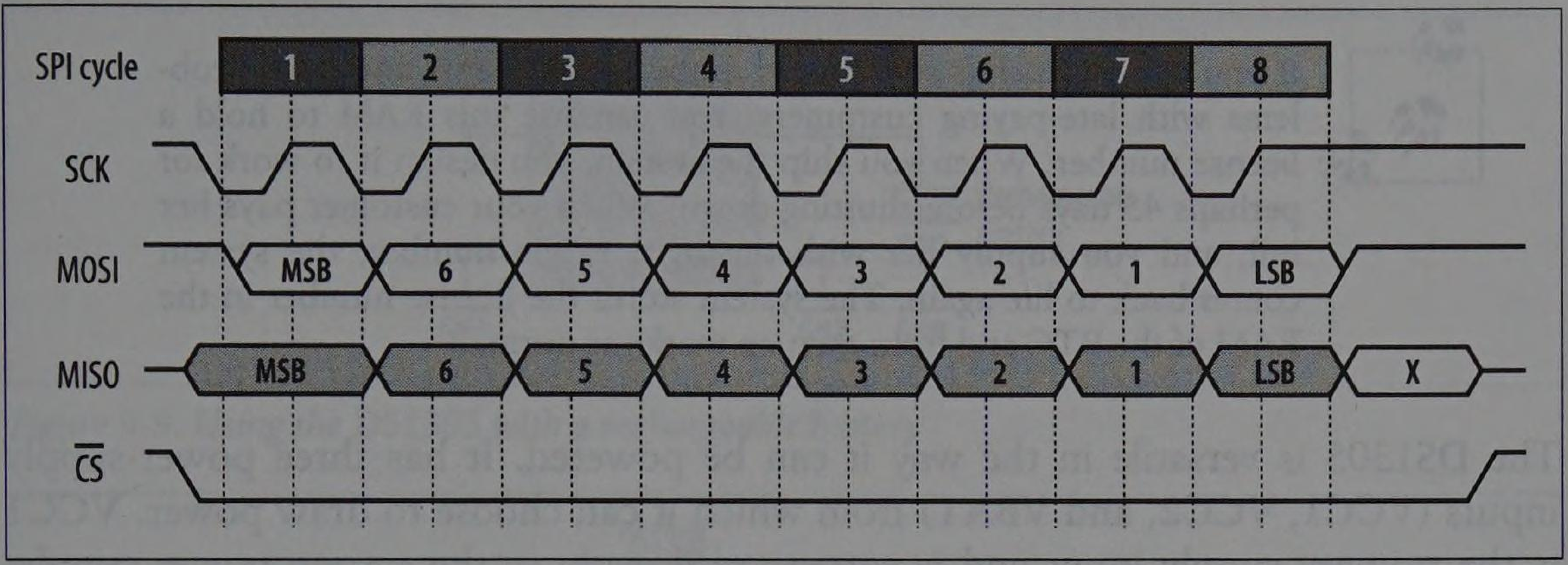


Figure 9-7. SPI timing with clock polarity high and clock phase one

minutes, hours, days, months, and years. They are specifically designed to provide accurate timekeeping, and many have additional functions such as an "alarm" (whereby the processor is interrupted at a specific time) and watchdog functions. Some also include voltage monitoring, so that the clock chip may act as a system monitor, alerting the processor should the power supply waver. A number of clock chips are available (and not all are interfaced using SPI). For this example, we will use the Maxim DS1305.

The way in which we interface the clock chip to a processor is virtually identical for all other SPI devices. Some chips with SPI interfaces have special requirements, but most are very simple and straightforward. This makes SPI a very useful interface that makes increasing system functionality trivial.

The Maxim DS1305 *Real-Time Clock* (*RTC*) provides timekeeping services and tracks seconds, minutes, hours, day of the month, month, day of the week, and year. It knows which months have 30 days and which have 31. It even automatically adjusts for leap years, up to the year 2100. It can generate two interrupts to the microcontroller for time-of-day alarms. These alarms can be used to trigger a regular system event, such as a backup or user notification.

The DS1305 can run off two separate power sources and supports battery backup of its internal state. The chip can use a power supply in the range 2V to 5.5V, allowing it to be powered from a variety of sources. It also has 96 bytes of static RAM, used for parameter storage. You could use the RAM for holding variables indicating system mode, secure password storage, or even authorization codes for your embedded software, just as desktop software does.

The RAM, like the timekeeping function, is battery backed, and so its contents will be retained for the life of the battery. This can be up to 10 years, depending on the battery chosen. Thus, the contents of the internal parameter RAM will probably last for the expected operational life span of an embedded system.

If you are producing commercial embedded systems and have problems with late-paying customers, you can use this RAM to hold a license number. When you ship the system, you design it to work for perhaps 45 days before shutting down. When your customer pays her bill, and you supply her with the right magic number, the system comes back to life again. The system stores the license number in the RAM of the RTC and from then on works as normal.

The DS1305 is versatile in the way it can be powered. It has three power-supply inputs (**VCC1**, **VCC2**, and **VBAT**) from which it can choose to draw power. **VCC1** is the primary supply input and is connected directly to the system power supply. When the computer is up and running, the DS1305 draws its current from this source. **VCC2** is the secondary power source, and this can be a rechargeable battery. **VBAT** is the third power source and is for nonrechargeable batteries.

There are three, and only three, possible configurations for powering the DS1305, and it is important for correct operation that the power inputs are appropriately driven. Figure 9-8 shows the DS1305 powered by a primary DC supply connected to **VCC1** and a secondary, nonrechargeable battery connected to **VBAT**. (To keep the diagram simple, only the power pins are shown. We'll look at the data interface in a moment.) For this configuration, **VCC2** is unused and must be connected to **GND**. When **VCC1** falls below a given threshold voltage (the primary power source has failed), the internal memory and registers of the DS1305 become write protected to prevent their being corrupted by a failing microprocessor.

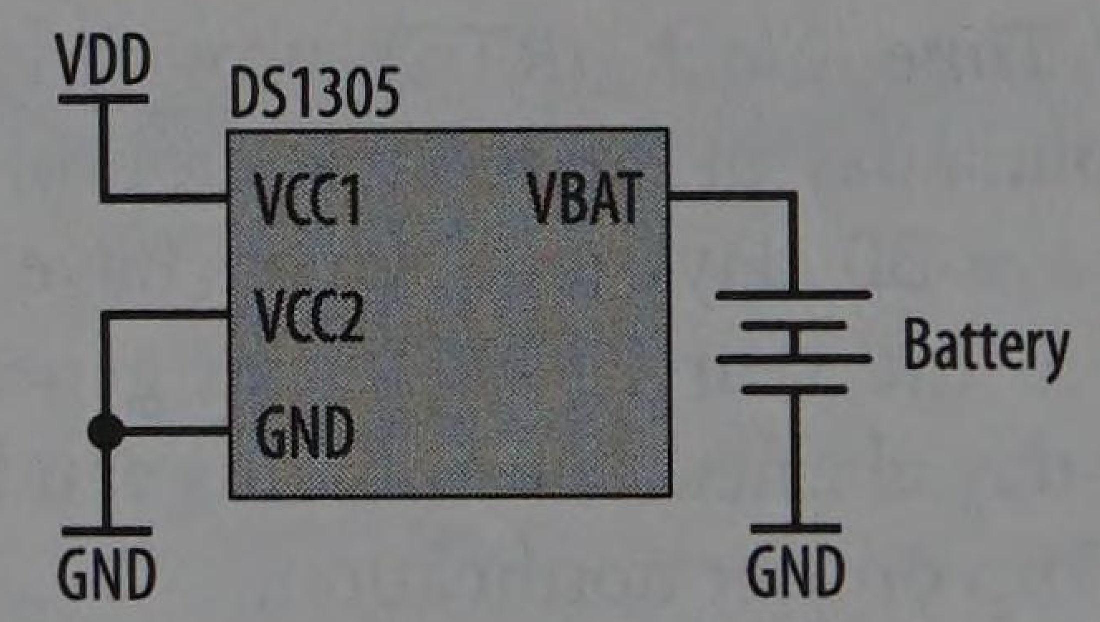


Figure 9-8. Using the DS1305 with a nonrechargeable battery

If the secondary power source is a rechargeable battery, then the DS1305 may be wired as shown in Figure 9-9. When using a rechargeable battery on **VCC2**, **VBAT** must be connected to **GND**. When the device is used in this mode, there is no automatic write protection for the DS1305 if **VCC1** fails.

Finally, the DS1305 may be used with only a battery as its primary power source and no backup power supply. This is shown in Figure 9-10. For this configuration, both **VCC1** and **VBAT** are connected to ground, while the battery is connected to **VCC2**.

Using the DS1305 is very simple. The schematic showing a DS1305 interfaced to a microcontroller is shown in Figure 9-11.

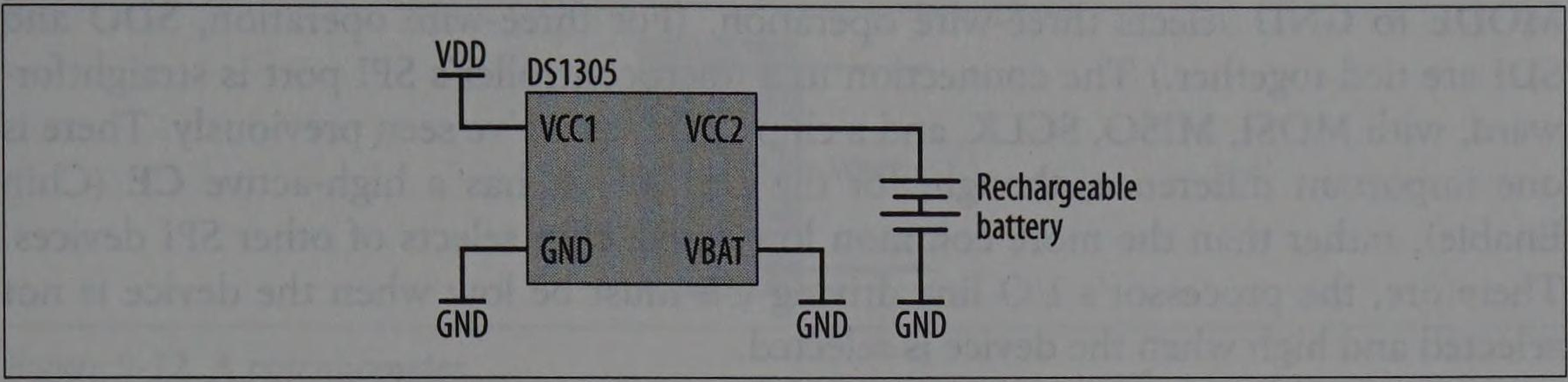


Figure 9-9. Using the DS1305 with a rechargeable battery

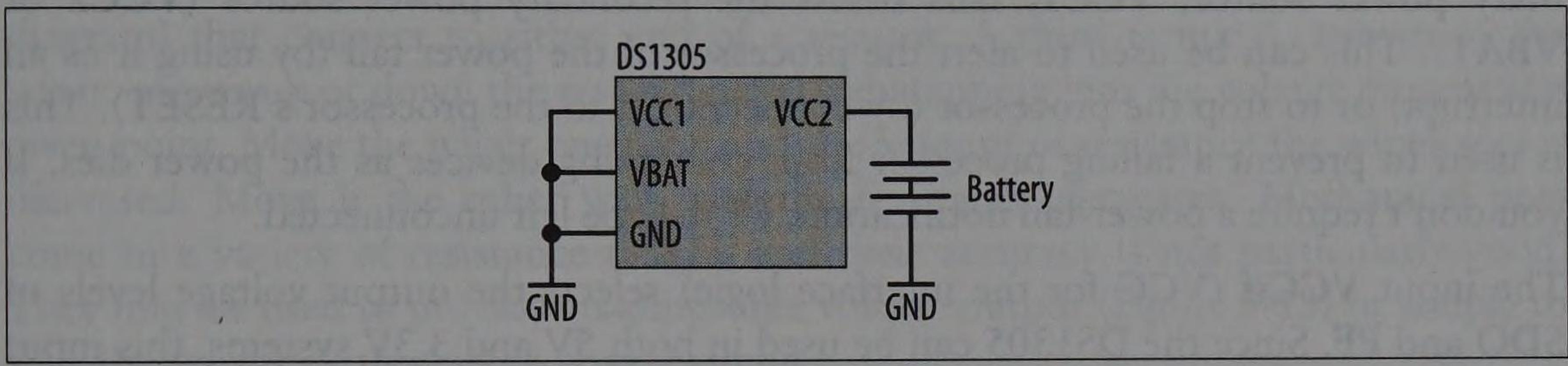


Figure 9-10. Using the DS1305 with a battery as its only power source

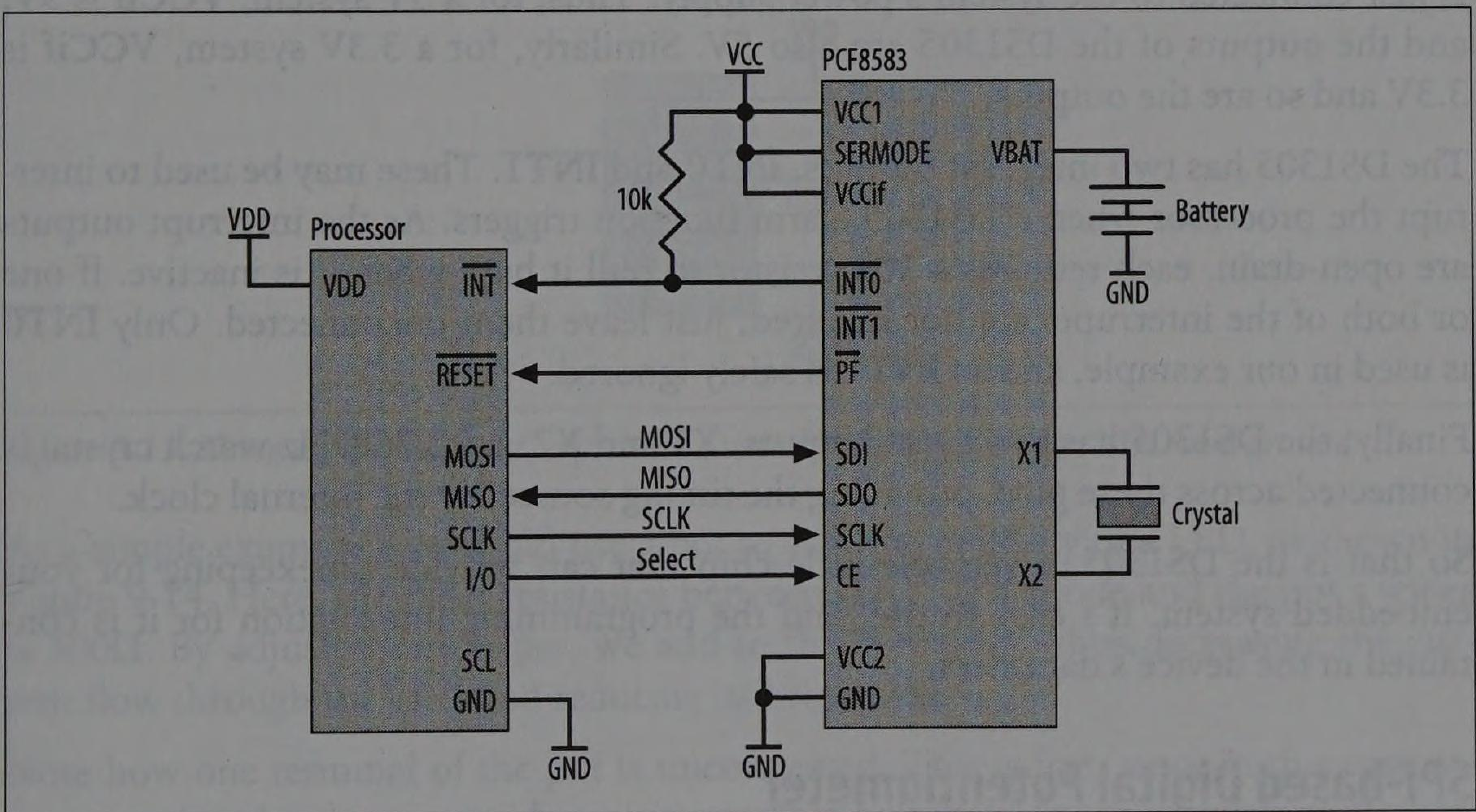


Figure 9-11. A DS1305 RTC interfaced to a microcontroller

The serial interface of the DS1305 can operate as either a SPI port or a three-wire* port. The input **SERMODE** (SERial MODE) selects which serial mode to use. Connecting **SERMODE** to the power supply selects SPI operation. Connecting **SER-**

* Developed by National Semiconductor, three-wire, also known as *MicroWire*, is very similar to SPI and is found in some microcontrollers and DSP processors. Unlike SPI, which has separate data lines for reading and writing, three-wire uses a common bidirectional data line.

MODE to **GND** selects three-wire operation. (For three-wire operation, **SDO** and **SDI** are tied together.) The connection to a microcontroller's SPI port is straightforward, with **MOSI**, **MISO**, **SCLK**, and a chip select, as we've seen previously. There is one important difference, though, for the DS1305. It has a high-active **CE** (Chip Enable), rather than the more common low-active chip selects of other SPI devices. Therefore, the processor's I/O line driving **CE** must be low when the device is not selected and high when the device is selected.

The DS1305 has a special Power Fail (**$\overline{\text{PF}}$**) output that is asserted low when the primary power source, **VCC1**, falls below the secondary power source (**VCC2** or **VBAT**). This can be used to alert the processor of the power fail (by using it as an interrupt) or to stop the processor (by connecting it to the processor's **$\overline{\text{RESET}}$**). This is used to prevent a failing processor from corrupting devices as the power dies. If you don't require a power-fail notification, **$\overline{\text{PF}}$** may be left unconnected.

The input **VCCif** (**VCC** for the interface logic) selects the output voltage levels of **SDO** and **$\overline{\text{PF}}$**. Since the DS1305 can be used in both 5V and 3.3V systems, this input allows the output levels of these pins to be set to the appropriate high voltage. **VCCif** is just connected to the system's power supply. Thus, for a 5V system, **VCCif** is 5V, and the outputs of the DS1305 are also 5V. Similarly, for a 3.3V system, **VCCif** is 3.3V and so are the outputs.

The DS1305 has two interrupt outputs, **$\overline{\text{INT0}}$** and **$\overline{\text{INT1}}$**. These may be used to interrupt the processor when a DS1305 alarm function triggers. As the interrupt outputs are open-drain, each requires a 10k resistor to pull it high when it is inactive. If one or both of the interrupts are not required, just leave them unconnected. Only **$\overline{\text{INT0}}$** is used in our example, and so **$\overline{\text{INT1}}$** is safely ignored.

Finally, the DS1305 has two crystal inputs, **X1** and **X2**. A 32.768kHz watch crystal is connected across these pins, providing the timing source for the internal clock.

So that is the DS1305, a versatile little chip that can provide timekeeping for your embedded system. It's easy to use, and the programming information for it is contained in the device's datasheet.

SPI-based Digital Potentiometer

Let's look at another simple SPI example. This time, we will interface a digital potentiometer to a microprocessor. Before getting into the details, let's take a look at what one is and why you'd use it. We'll get back into SPI in just a moment.

A *potentiometer* (also known simply as a *pot*, *trimmer*, or *trim pot*) is just a variable resistor. The symbol for a pot is shown in Figure 9-12. Pots are normally mechanical components and are manually adjusted. Your stereo probably uses pots for its volume, bass, and treble controls. The brightness and contrast knobs for monitors and LCDs are also potentiometers.

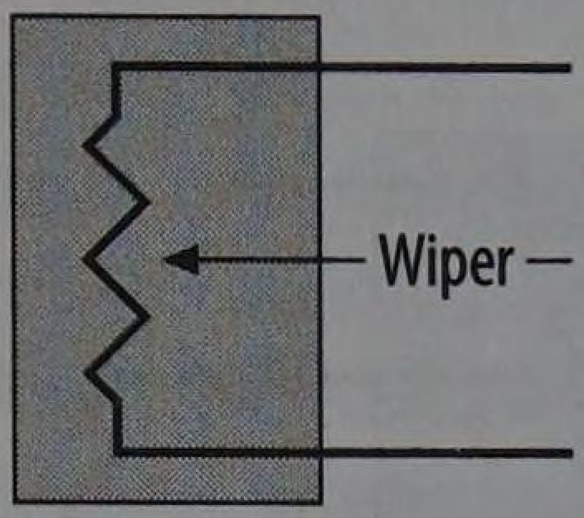


Figure 9-12. A potentiometer

A standard potentiometer consists of two terminals (the upper and lower pins in the diagram) that connect to either end of a resistor. A third terminal, known as the *wiper*, moves up or down the resistor, effectively tapping into the voltage present at a given point. Move the wiper one way, and the amount of resistance the wiper sees is increased. Move it the other way, and the resistance decreases. Mechanical pots come in a variety of resistance ranges, and their accuracy is not particularly good. They may be used to provide an adjustable voltage output (Figure 9-13) or simply to vary the resistance used in an analog circuit.

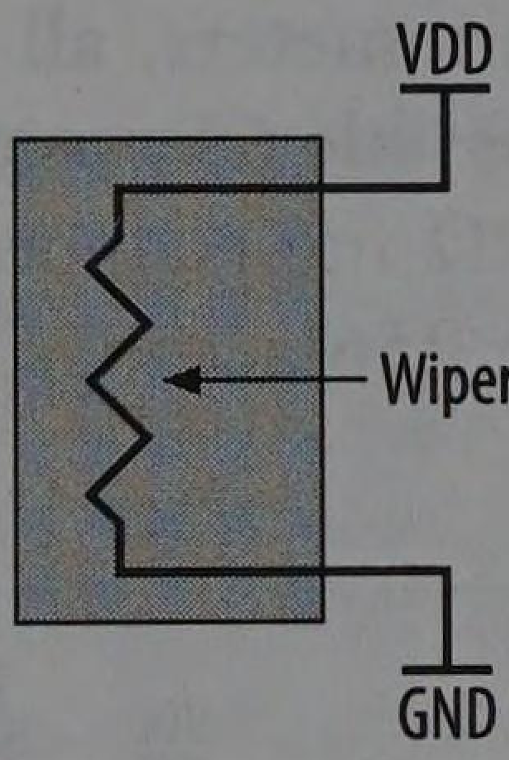


Figure 9-13. Using a potentiometer to provide a variable voltage between VDD and ground

As a simple example, you could use a pot to vary the intensity of a LED, as shown in Figure 9-14. Here, the fixed resistance between the LED's anode and the pot's wiper is 300Ω. By adjusting the wiper, we add to this resistance, thus decreasing the current flow through the LED and reducing its brightness.

Note how one terminal of the pot is unconnected. This is fine, since in this case we are not using the pot to provide an intermediate voltage between two values. Rather, we are simply using the pot as a variable resistor, increasing the impedance between the wiper and **VDD**.

Now, a standard pot is manually adjusted. It either will have a knob attached (as in a volume control or brightness adjustment) or will have a small notch for screwdriver adjustment. Wouldn't it be great if your microprocessor could adjust the pots in your analog circuits, under software control? That way, your application software could adjust the brightness of the display or change the volume of the sound system. Well, by using a digital potentiometer, you can do just that. Televisions, computer monitors, and stereos with internal embedded controllers use digital pots to adjust

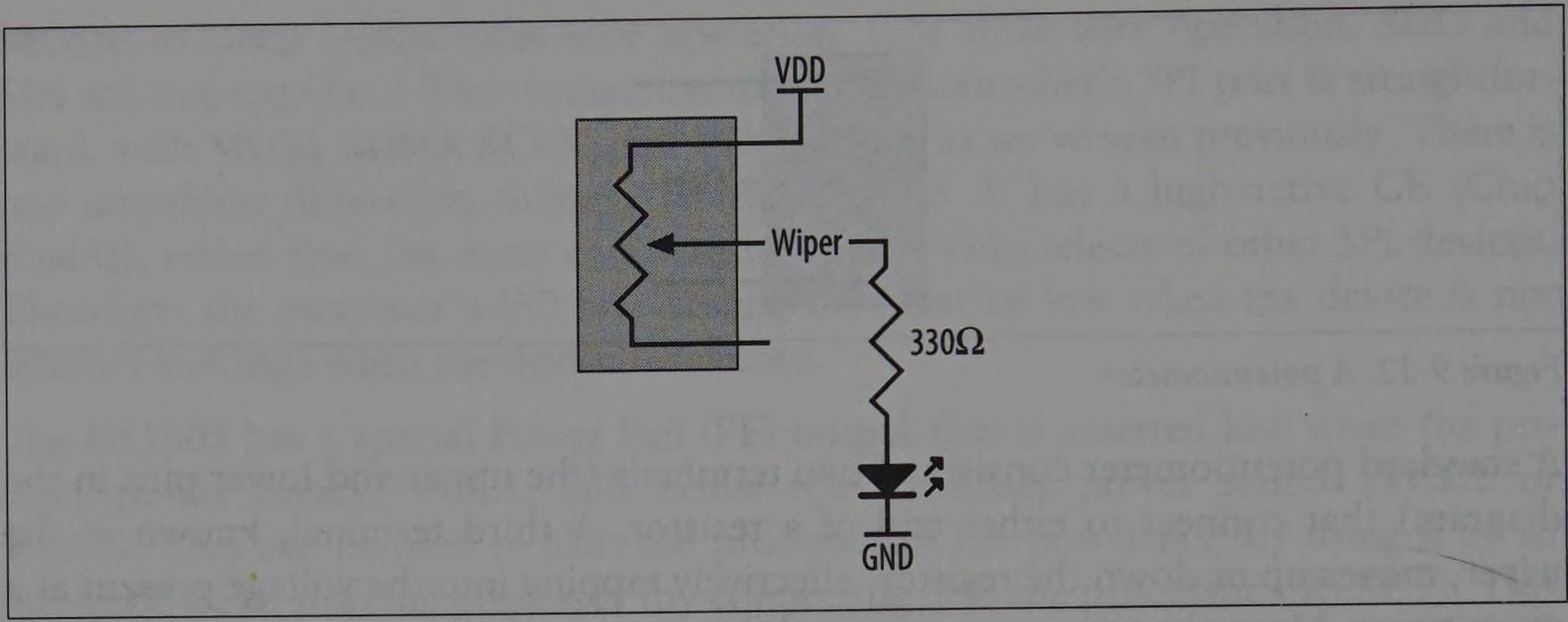


Figure 9-14. Using a potentiometer to vary the intensity of a LED

settings such as volume. When you hit a volume button on a remote control, the TV or stereo adjusts the settings of digital pots, which are part of the amplifiers driving the speakers.

Figure 9-15 shows an Analog Devices AD5203 digital potentiometer with a SPI interface. This chip has four potentiometers, all of which may be adjusted under software control. Each pot has a possible 64 positions, and versions of the chip are available with either 10kΩ or 100kΩ impedances. For higher resolution, the pin-compatible AD8403 has a possible 256 settings, also configurable through a SPI interface.

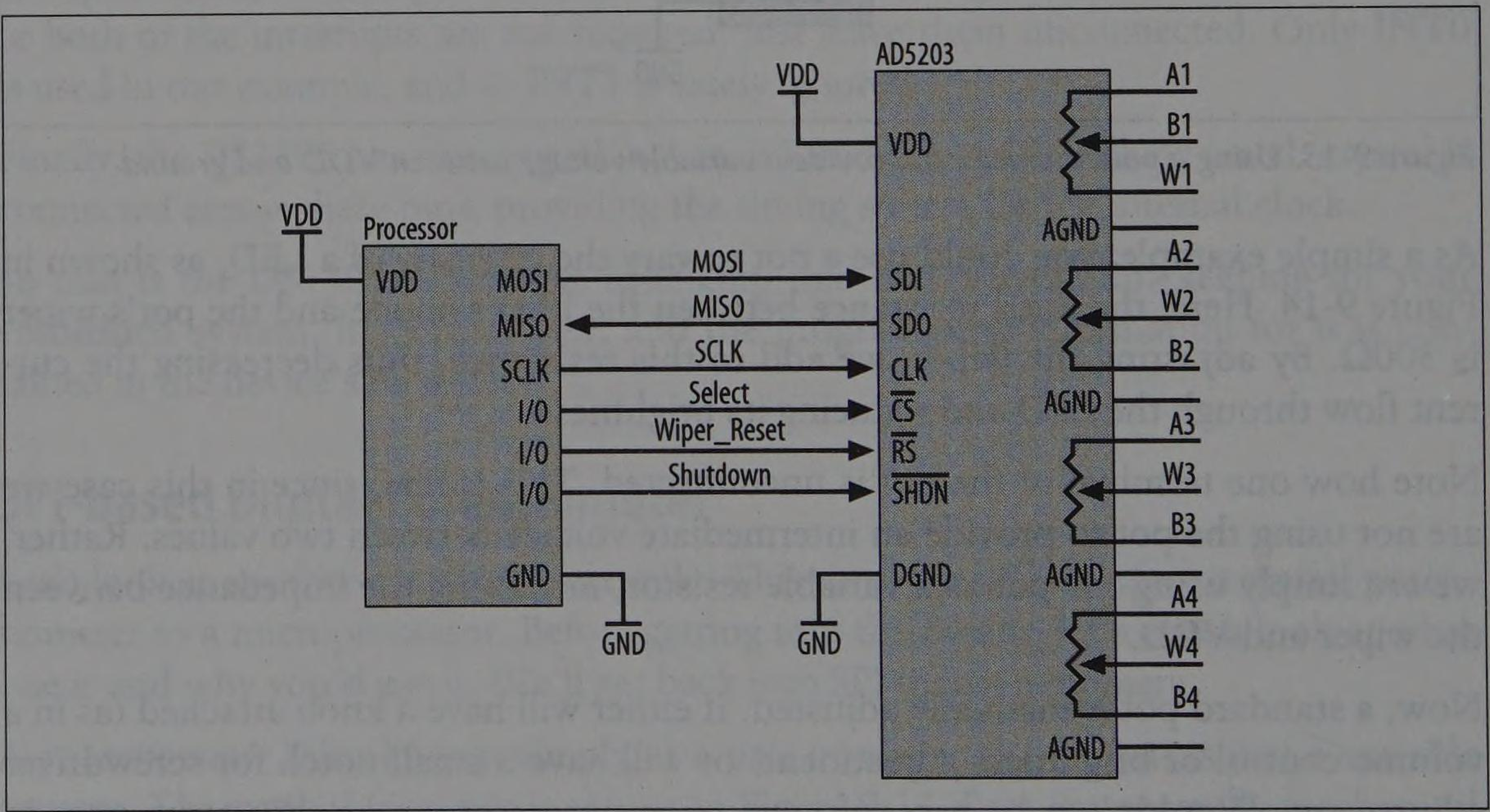


Figure 9-15. Interfacing a digital potentiometer to a processor using SPI

The AD5203 has a Serial Data Input (**SDI**), which is connected to the processor's **MOSI** output. Similarly, the device's Serial Data Output (**SDO**) is connected to

MISO. The AD5203's clock input (**CLK**) is positive-edge-triggered midway through each SPI cycle, which means that any processor communicating with it must use high clock polarity and clock phase one on **SCLK**. The Chip Select ($\overline{\mathbf{CS}}$) of the AD5203 may be driven by a processor digital I/O line. The AD5203 has two other inputs, Shutdown ($\overline{\mathbf{SHDN}}$) and Reset ($\overline{\mathbf{RS}}$). $\overline{\mathbf{SHDN}}$ places the device in low-power mode, and $\overline{\mathbf{RS}}$ resets the potentiometer wipers to their midpoint position. Both of these inputs may also be driven by a processor I/O line, or if their functionality is not needed, they may be simply tied high using 10kΩ pull-up resistors.

The potentiometers within the AD5203 are used as any other pots would be. The **A** and **B** terminals connect to either end of the internal resistors, and the position of the wiper (**W**) is adjusted under software control.

The AD5203 has several ground connections. **DGND** is the digital ground for the SPI interface and control logic of the chip. **AGND**s are the analog grounds of the internal potentiometers and should all be connected to **DGND** at a single point.

The datasheet for the AD5203 provides the control codes needed to configure the chip, and its use is simple and straightforward.

Adding Nonvolatile Data Memory with SPI

The internal memory of microcontrollers is very small, and their data storage capabilities are severely limited. We're now going to look at how you can increase the storage capacity of your embedded system by adding an ATMEL AT45DB161 2M serial DataFlash using SPI. These chips are commonly used in low-cost digital cameras and answering machines. You could also use this flash chip as a virtual disk drive in your embedded system.

Most other flash chips have a bus interface, but the AT45DB161 has a serial interface, making it well-suited for use with small microcontrollers. The AT45DB161 is a 2M chip, but you can get similar ATMEL chips in capacities ranging from 512K to 8M. They all use the same physical interface, so the same design works for all. (Note, however, that their pinouts and physical packages vary, so one chip will not mount onto a circuit board designed for another.)

The chip consists of an array of flash memory, organized as individual pages of 528 bytes each, and two RAM buffers, also 528 bytes each (Figure 9-16). To write data into the main flash array, the processor must first write data into one of the buffers and then issue a command to write that buffer into the array. A processor can read the contents of either of the buffers, transfer a flash page to the buffers, or read from the flash array directly. The operation of the buffers is independent, and one buffer may be accessed by the processor (via SPI) while the contents of the other buffer are being written into the flash array.

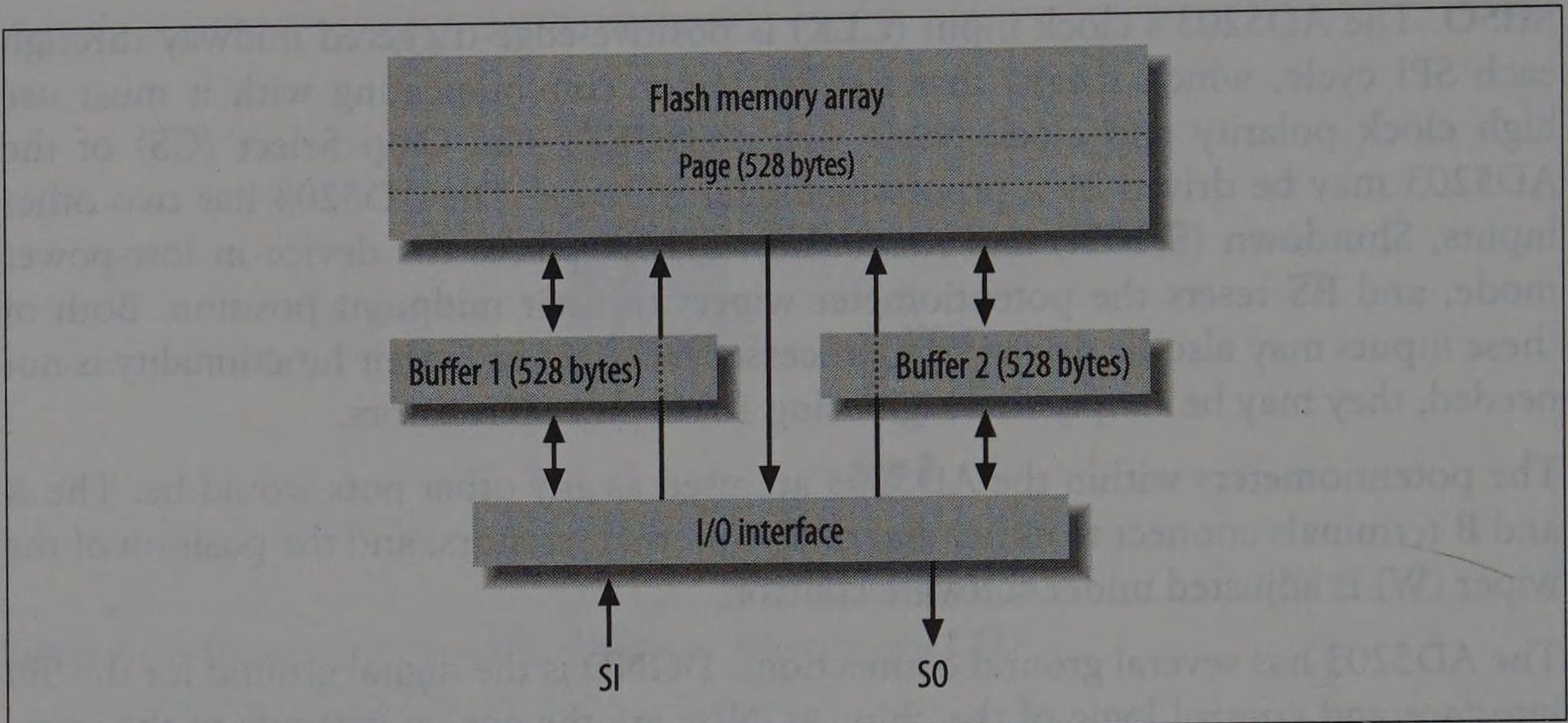


Figure 9-16. AT45DB161 internal architecture

The flash supports numerous commands for writing to and reading from the buffers, writing the buffers to the main array and transferring an array page back to a buffer. The ATMEL datasheet has full details of the software protocols and command set.

There are a few things to note about the internal architecture and the flash array. The first is that one 528-byte page of the flash array is not contiguous with the next. In other words, if you are using a pointer in your software to track the current location in the memory, you can't just increment it from the end of one page and expect it to be pointing to the next. Every 528 bytes (and it's a strange number), you have to leap forward to the next page. Think of it as pages of 528 bytes with big gaps in between.

The second catch with this memory is that it only has a lifetime of 1000 write cycles per page. Most flash technologies (and there are several different types) support 100,000 write cycles or better, and you can normally exceed this limit and the device will keep working reliably for you. This isn't the case with the AT45DB161. Once the 1000-write limit is exceeded, memory locations start failing on you. The chip will read existing data back correctly, but new pages will not write successfully. Depending on the application, this limit may not be a problem. My company uses this particular chip in our long-term dataloggers. These machines are deployed for year-long deployments, collecting (and compressing) data and storing it away in the flash chip. The logger gradually builds a page image in one of the buffers before storing it to the array in a single write. Since, during a deployment, a page will be written only once (and then the logger will move on to the next page), the 1000-write limitation isn't a problem. It would take a thousand deployments before the chip would fail. However, if you are using the chip for variable storage and are modifying the flash pages on a byte-by-byte basis, you're in trouble. Individually changing 528 bytes within a page counts as 528 writes. So do that twice to a page, and suddenly you're over the limit. Therefore, this flash is well-suited to some applications and not others.

The basic design for using an AT45DB161 is shown in Figure 9-17.

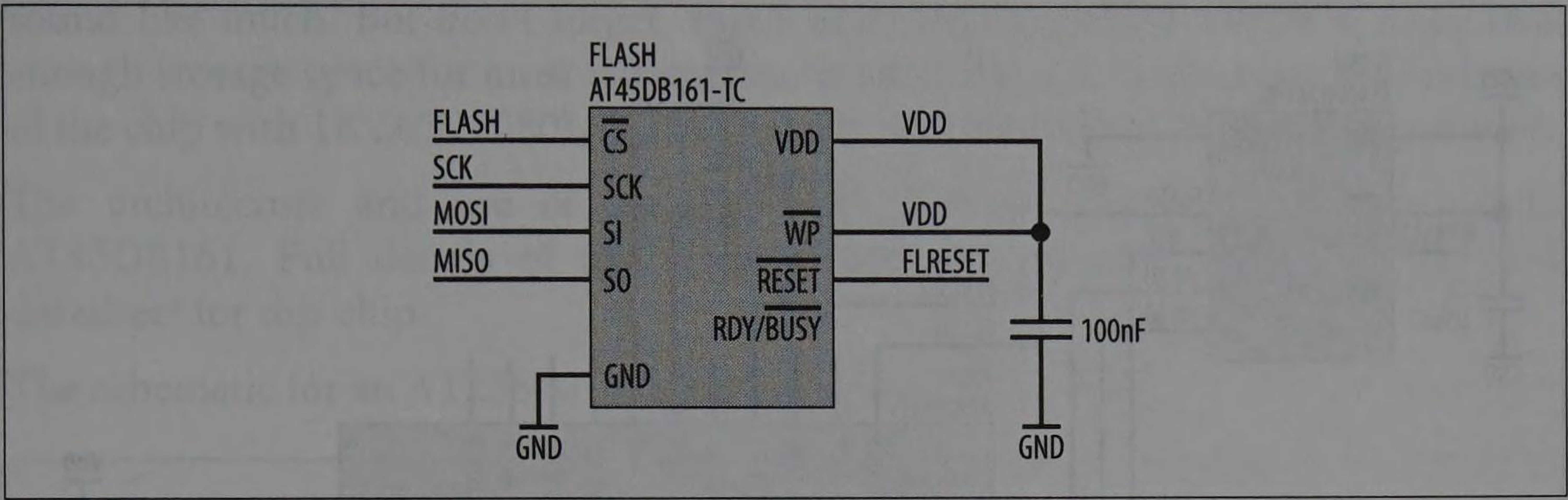


Figure 9-17. 2M serial DataFlash

On the left of the chip are the SPI interface connections, **MOSI**, **MISO**, and **SCK**, and a chip select (**FLASH**). The chip will support SPI transfers at up to 20MHz, so the SPI interface can be run very fast indeed. On the right of the chip is the power supply, **VDD**, which is decoupled to ground using a 100nF capacitor. The AT45DB161 requires a power supply in the range of 2.5V to 3.6V. However, its logic inputs are 5V tolerant, meaning that this chip can be used in systems with mixed power supplies. In other words, while this chip requires a 3V power supply, it can be directly interfaced to a processor with a 5V supply (and 5V logic levels). The AT45DB161 has a write-protect pin (**WP**), which, when driven low, prevents the contents of the flash being modified. If you don't require write protection, simply tie this input high, as shown in the schematic. The flash also has a **RESET** input so that the chip can be manually reset under software control. The flash incorporates a built-in power-on reset that will put the device into a known state; therefore, a "manual" reset at power-up should be unnecessary. However, I've found that the internal power-on reset generator is somewhat finicky and doesn't always kick in as it should. Under such circumstances, the flash fails to enter a known state and is unusable in the system. Therefore, I have found it good practice to give the processor control of the flash's reset. As part of the processor initialization routines executed in its reset firmware, I get the processor to reset the flash, nudging it into reality. It's a simple thing, but makes all the difference for a reliable system. Pin 1 is a status output (**RDY/BUSY**) indicating whether the device is ready or if it is still completing an internal operation.

The connections for interfacing this memory chip to an ATMEL 90S4434 AVR processor are shown in Figure 9-18. The AVR portion of the schematic is no different from the examples we have seen previously. That's the nice thing about simple interfaces such as SPI. They form little subsystem modules that "bolt together" like building blocks. Start with the basic core design, and just add peripherals as you need them. The schematic also shows decoupling capacitors for the power supplies, the crystal oscillator for the processor, and a pull-up resistor for **RESET**. Pin 41 (**PB1**) is used as a "manual" (processor-controlled) reset input to the flash.

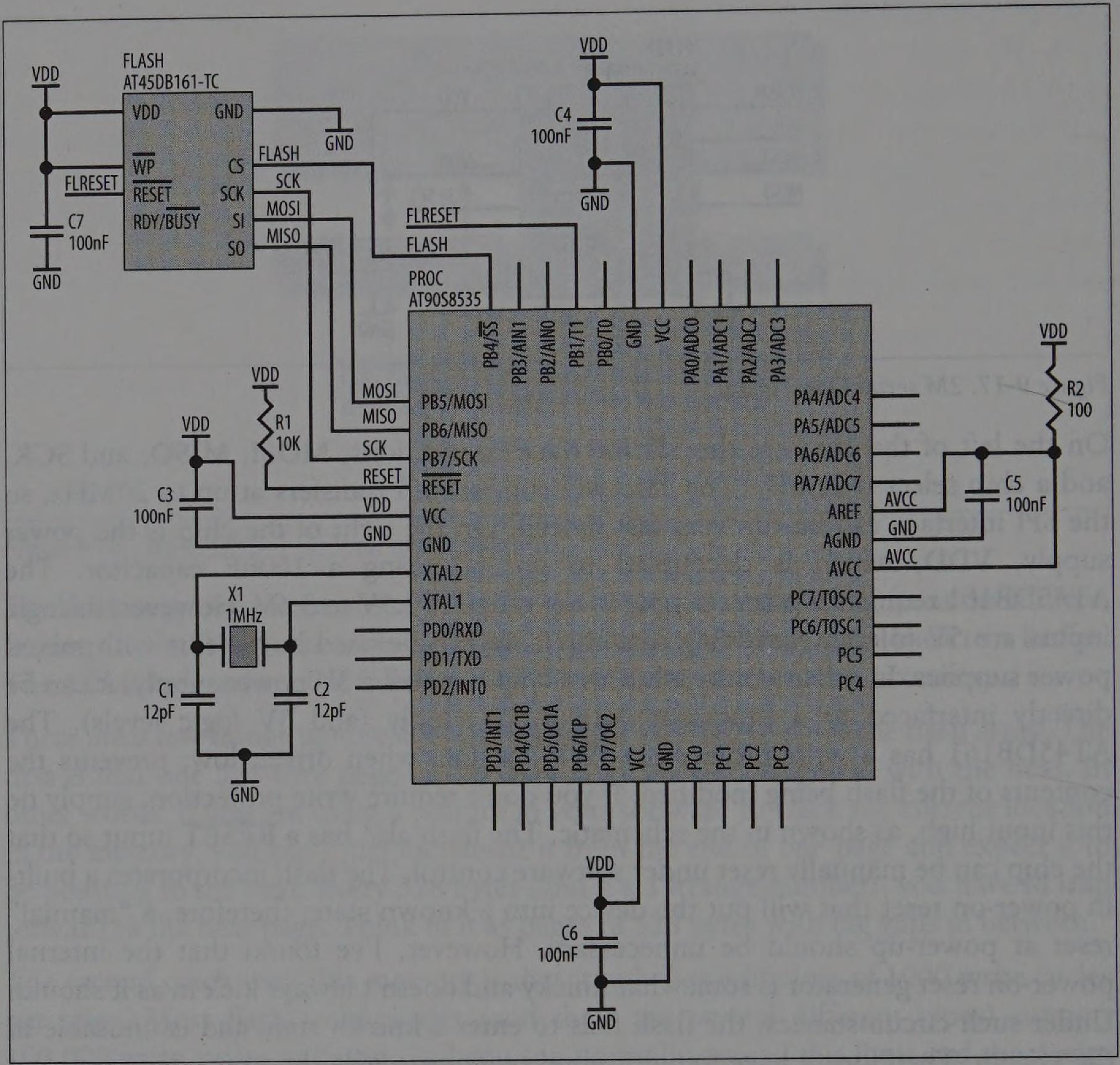


Figure 9-18. A 2M DataFlash interfaced to an AT90S4434

Adding a Parameter Memory Using SPI

We saw in the previous section how to add a large-capacity serial flash for data storage. Using nonvolatile memory to hold system parameters can be valuable as a way of preserving important key variables during periods of no power. But the AT45DB161 DataFlash is just not the device for that task. It is better suited to data recording, and its large capacity is overkill for parameter storage. So, now we're going to look at how you can use SPI to add a small parameter memory (in the form of an EEPROM) to your embedded system. The EEPROM I've chosen is the ATMEL AT25640. This device will hold data for at least 100 years without power and will endure more than one million write cycles (significantly more than an AT45DB161!). In that way, your software can happily alter parameter variables without fear of limiting the life span of the chip. The AT25640 has only 8K of memory, which might not

sound like much. But don't forget, that's 8192 char variables, which is more than enough storage space for most parameters. If 8K is too much, there are also versions of the chip with 1K (AT25080), 2K (AT25160), and 4K (AT25320) bytes of memory.

The architecture and use of the AT25640 is much simpler than that of the AT45DB161. Full details of the required software protocol are in the ATMEL datasheet for this chip.

The schematic for an AT25640 circuit is shown in Figure 9-19.

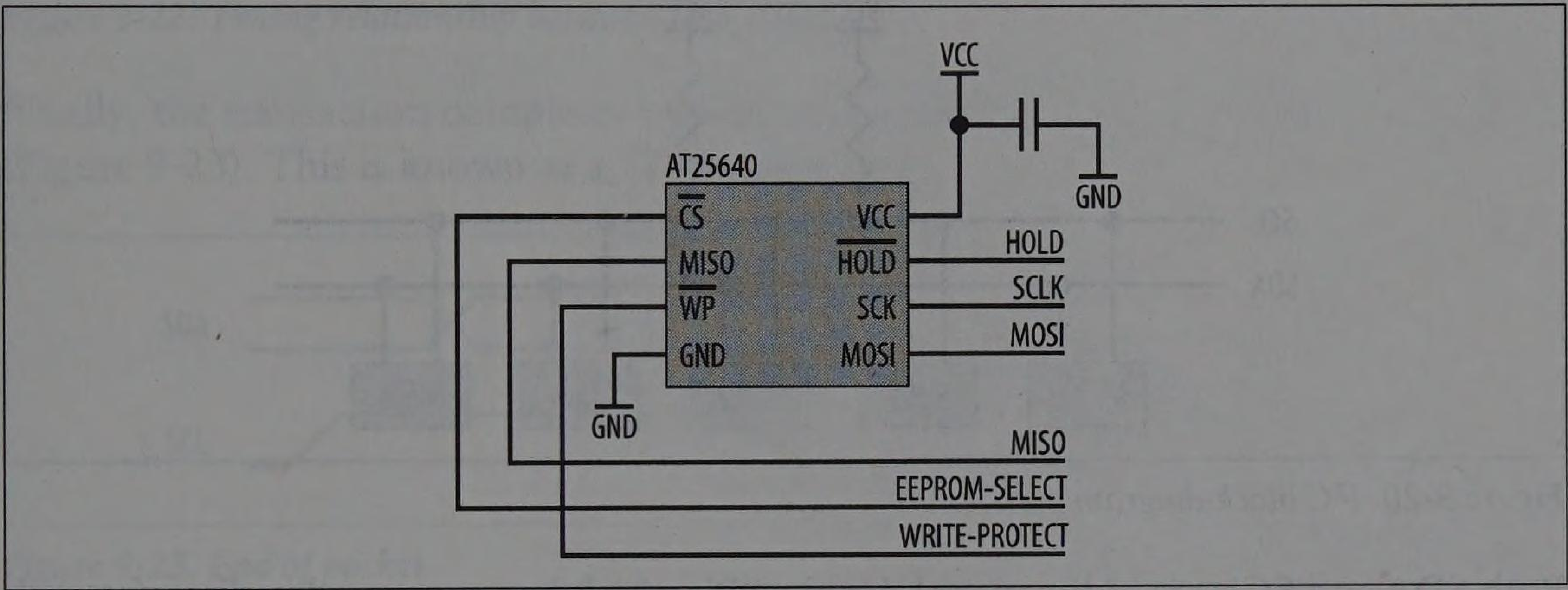


Figure 9-19. Using an AT25640 EEPROM

The interface is standard SPI, and the chip also has a write-protect input and a hold input. Asserting hold allows the processor to temporarily stall a serial transfer (while it performs other tasks) without terminating the access to the AT25640. And, as you might expect, write-protect, when asserted, turns the chip into a read-only device. These control inputs may be driven by programmable I/O lines of the processor. The only other requirement is power (which is decoupled to ground using a 100nF capacitor) and ground. The chip is available in two types. One will operate from a supply voltage of between 2.7V and 5.5V, while the other needs a supply voltage of between 1.8V and 3.6V.

Inter Integrated Circuit

The I^2C (*Inter Integrated Circuit*) bus is a very cheap, yet effective, network used to interconnect peripheral devices within small-scale embedded systems. It is sometimes also known as *IIC* and has been in existence for more than 20 years. It is the equivalent of SPI, but its operation is somewhat different.

I^2C uses two wires to connect multiple devices in a multidrop bus. The bus is bidirectional, low-speed, and synchronous to a common clock. Devices may be attached or detached from the I^2C bus without affecting other devices. Several manufacturers, such as Microchip, Philips, Intel, and others produce small microcontrollers with I^2C built in. The data rate of I^2C is somewhat slower than SPI, at 100kbps in standard mode and 400kbps in fast mode.

The two wires used to interconnect with I²C are **SDA** (serial data) and **SCL** (serial clock). Both lines are open-drain.* They are connected to a positive supply via a pull-up resistor and therefore remain high when not in use. A device using the I²C bus to communicate drives the lines low or leaves them pulled high as appropriate. Each device connected to the I²C bus has a unique address and can operate as a transmitter (a bus master), a receiver (a bus slave), or both (Figure 9-20). I²C is a *multimaster bus*, meaning that more than one device may assume the role of bus master.

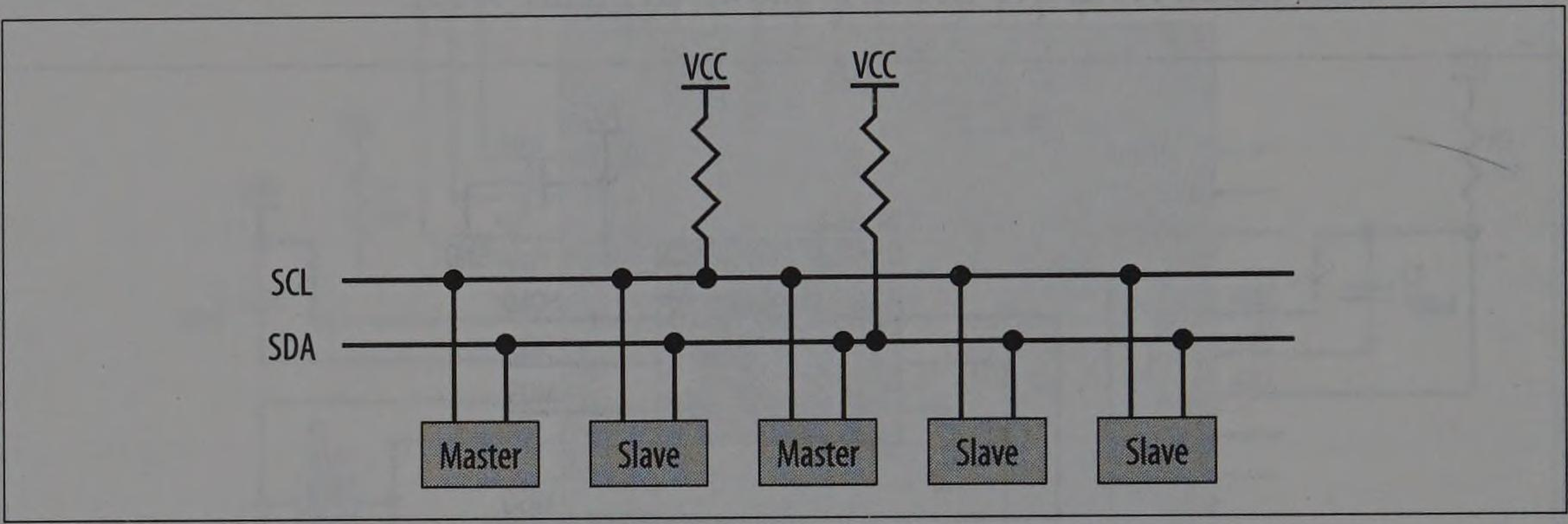


Figure 9-20. I²C block diagram

Both **SDA** and **SCL** are bidirectional. Unlike SPI, which has separate data lines for each direction of communication, I²C shares the same signal line for master transmission and slave response. Also unlike SPI, I²C does not have several modes of operation. The timing relationship between the clock, **SCL**, and the data line, **SDA**, is simple and straightforward. When idle, both **SDA** and **SCL** are high. An I²C transaction begins with **SDA** going low, followed by **SCL** (Figure 9-21). This indicates to all receivers on the bus that a packet transmission is commencing. While **SCL** is low, **SDA** transitions (high or low) for the first valid data bit. This is known as a *START condition*.

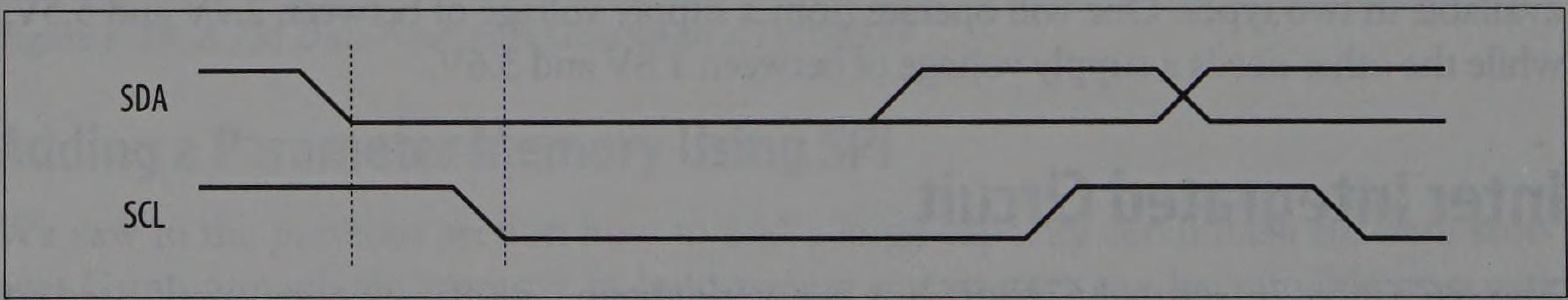


Figure 9-21. Start of packet

For each bit that is transmitted, the bit must become valid on **SDA** while **SCL** is low. The bit is sampled on the rising edge of **SCL** and must remain valid until **SCL** goes

* An *open-drain* or *open-collector* pin has output drivers that can only pull the signal line to ground. They cannot drive it high. This has the advantage that more than one device connected to a signal line may pull it low. If this were not the case, one device attempting to pull the line low while another tried to pull it high would result in a short circuit, with disastrous results. Interrupt lines are typically open-collector. All open-collector signals need a pull-up resistor and are low active. The idle state (when no device is asserting) is to be pulled high by the resistor.

low once more. Then **SDA** transitions to the next bit, before **SCL** goes high once more (Figure 9-22).

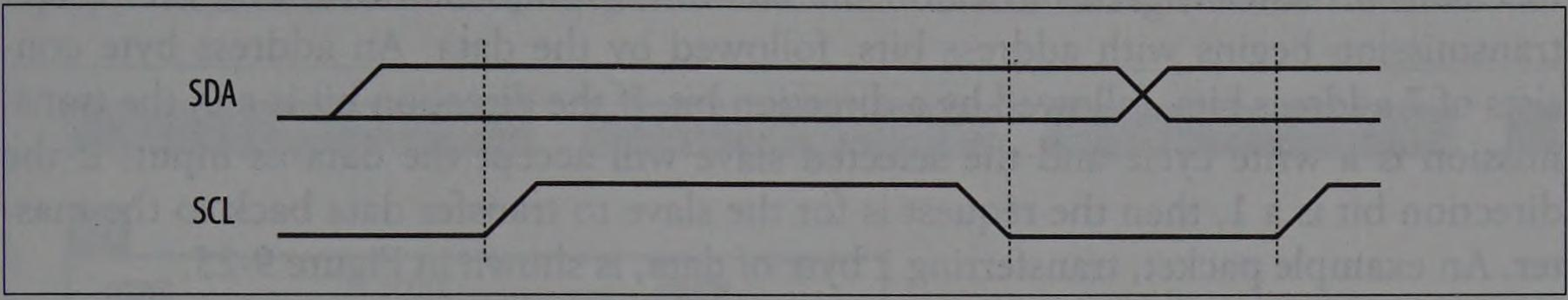


Figure 9-22. Timing relationship between SDA and SCL

Finally, the transaction completes by **SCL** returning high (inactive) followed by **SDA** (Figure 9-23). This is known as a *STOP condition*.

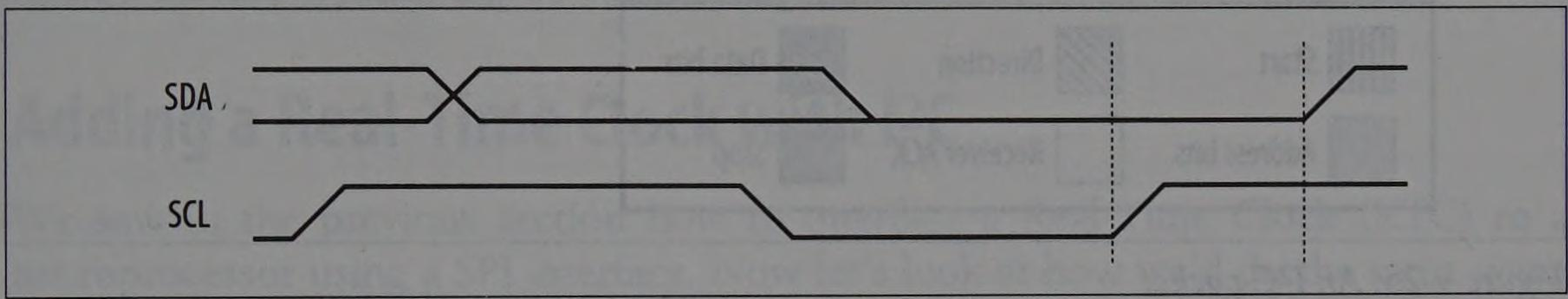


Figure 9-23. End of packet

Any number of bytes may be transmitted in an I²C packet. As with SPI, the most significant bit of the packet is transmitted first. If the receiver is unable to accept any more bytes, it can abort the transmission by holding **SCL** low. This forces the transmitter to wait until **SCL** is released again.

Each byte transmitted must be acknowledged by the receiver. Upon the transmission of the eighth data bit, the master releases the data line **SDA**. The master then generates an additional clock pulse on **SCL**. This triggers the receiver to acknowledge the byte by pulling **SDA** low (Figure 9-24). If the receiver fails to pull **SDA** low, the master aborts the transfer and takes appropriate error-handling measures.

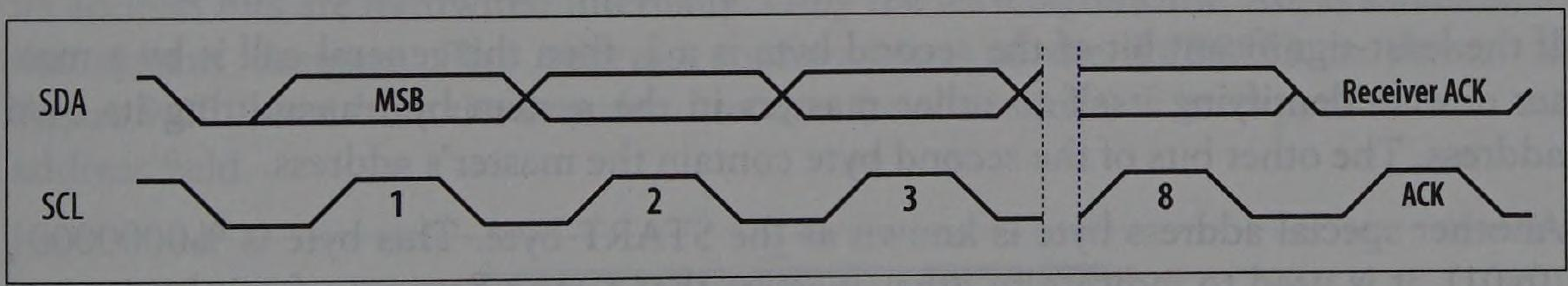


Figure 9-24. I²C packet with receiver acknowledge

Now, I²C is a multimaster bus. So, more than one master may attempt to start transmission at the same time. Since the bus's default state is high, a master transmitting a 0 bit will pull **SDA** low but will leave the bus in its default state if the bit is to be a 1. Thus, if two masters begin simultaneous transmission, a master leaving the bus in its default state for a 1 bit, but detecting the bus pulled low by another master (for a 0 bit), will register an error condition and abort the transmission.

Now, SPI used a separate chip select to enable a receiving slave. Each SPI slave has a separate chip select, generated by the master. I²C does not have such a selection mechanism. Instead, each device on the bus has a unique address, and the packet transmission begins with address bits, followed by the data. An address byte consists of 7 address bits, followed by a direction bit. If the direction bit is a 0, the transmission is a write cycle and the selected slave will accept the data as input. If the direction bit is a 1, then the request is for the slave to transfer data back to the master. An example packet, transferring 1 byte of data, is shown in Figure 9-25.

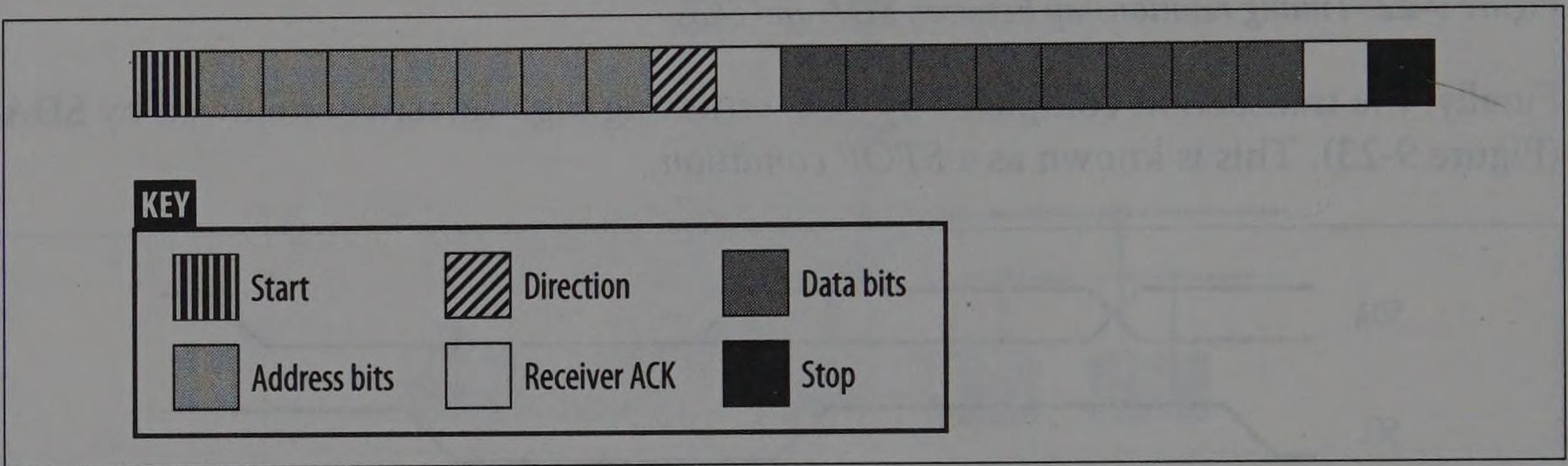


Figure 9-25. An I²C packet

A special address, known as the *general call address*, broadcasts to all I²C devices. This address is %0000000 with a direction bit 0. The general call is the mechanism by which the master determines what slaves are available, and there are several types of general call. The second byte of a general call indicates the purpose of the general call to the slaves. Upon receiving the second byte, individual slaves will determine whether the command is applicable to them, and if so they will acknowledge. If the command is not applicable to a given slave, then the slave simply ignores the general call and does not acknowledge. If the second byte is 0x06 (%00000110), then this indicates that appropriate slaves should reset and respond with their addresses. If the second byte is 0x04 (%00000100), slaves respond with their addresses but do not reset. Any other second byte of a general call, where the least-significant bit is a 0, should be ignored.

If the least-significant bit of the second byte is a 1, then the general call is by a master device identifying itself to other masters in the system by transmitting its own address. The other bits of the second byte contain the master's address.

Another special address byte is known as the START byte. This byte is %00000001 (0x01). It is used to indicate to other masters that a long data transfer is beginning. This is particularly important for masters that do not have dedicated I²C hardware and must monitor the bus by software polling. When a master detects a START byte generated by another master, it can reduce its polling rate, allowing it more time for other software tasks.

I²C also supports an extended 10-bit addressing mode, allowing up to 1024 peripherals. Devices that use 7-bit addressing may be mixed with 10-bit addressing devices in a single system. In 10-bit addressing, 2 bytes are used to hold the address. If the

(first) address byte begins with %11110XX, then a 10-bit address is being generated. The 2 least-significant bits of the first byte, combined with the 8 bits of the second byte, form the 10-bit address (Figure 9-26); 7-bit devices will ignore the transaction.

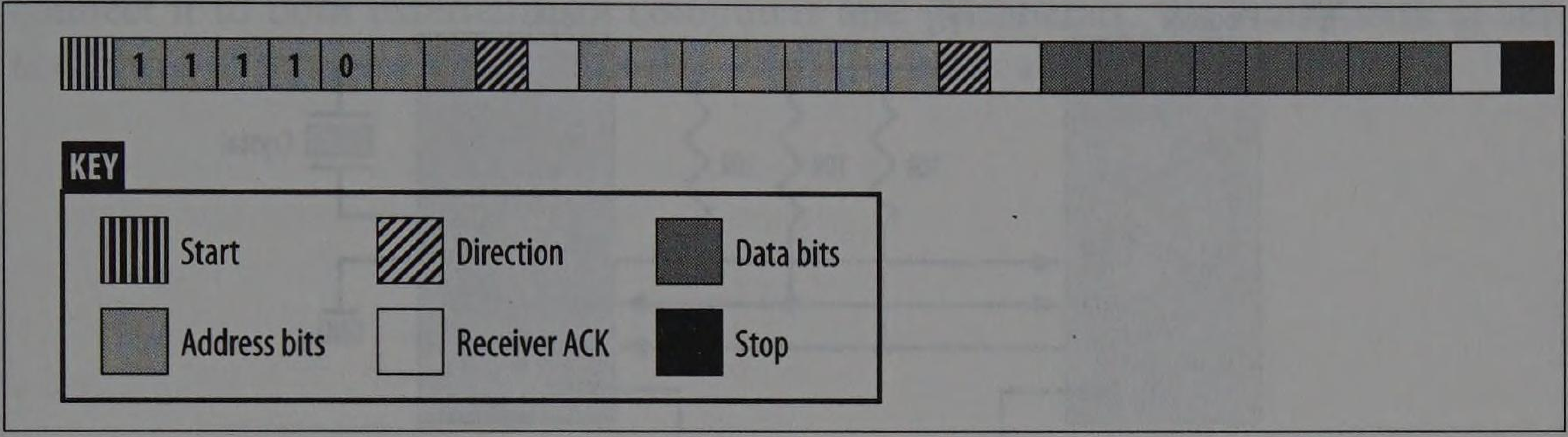


Figure 9-26. An I²C packet with 10-bit addressing

Adding a Real-Time Clock with I²C

We saw in the previous section how to interface a Real-Time Clock (RTC) to a microprocessor using a SPI interface. Now let's look at how we'd do the same using the I²C interface. For this example, we'll use the tiny Philips PCF8583. It also has 240 bytes of RAM, which, like the DS1305's, may be used for parameter storage. Unlike the DS1305, the PCF8583 does not have an integrated battery-backup system. So, you would need to provide an external battery-backup circuit. Many other I²C RTCs are available, and some do incorporate battery-fail protection. I've chosen to look at this one because it makes for a very simple example of an I²C interface.

The PCF8583 has two pins (**OSCI** and **OSCO**) for connecting a 32.768kHz watch crystal. This crystal pulses an internal circuit that performs the timekeeping functions. The address pin, **A0**, determines the address of the device on the I²C bus. Most I²C chips provide several address pins, allowing a range of possible addresses to be wired. The PCF8583 has only one, to reduce the pin count of the chip. Six of its address bits are hardwired internally. Only the least significant, **A0**, is available to the system designer. The address configuration of the PCF8583 is shown in Figure 9-27. Note how the transfer direction (read or write) is incorporated into the address field.

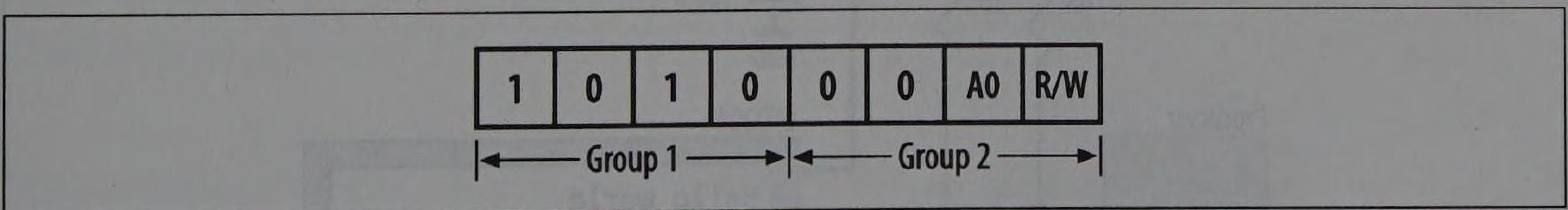


Figure 9-27. PCF8583 addresses

Connecting **A0** directly to ground sets that address bit to 0 and therefore maps the PCF8583 to I²C address 0x50. Alternatively, if **A0** is tied to **VDD**, then the address of the device is 0x51.

The schematic for interfacing the PCF8583 to a microcontroller is shown in Figure 9-28.

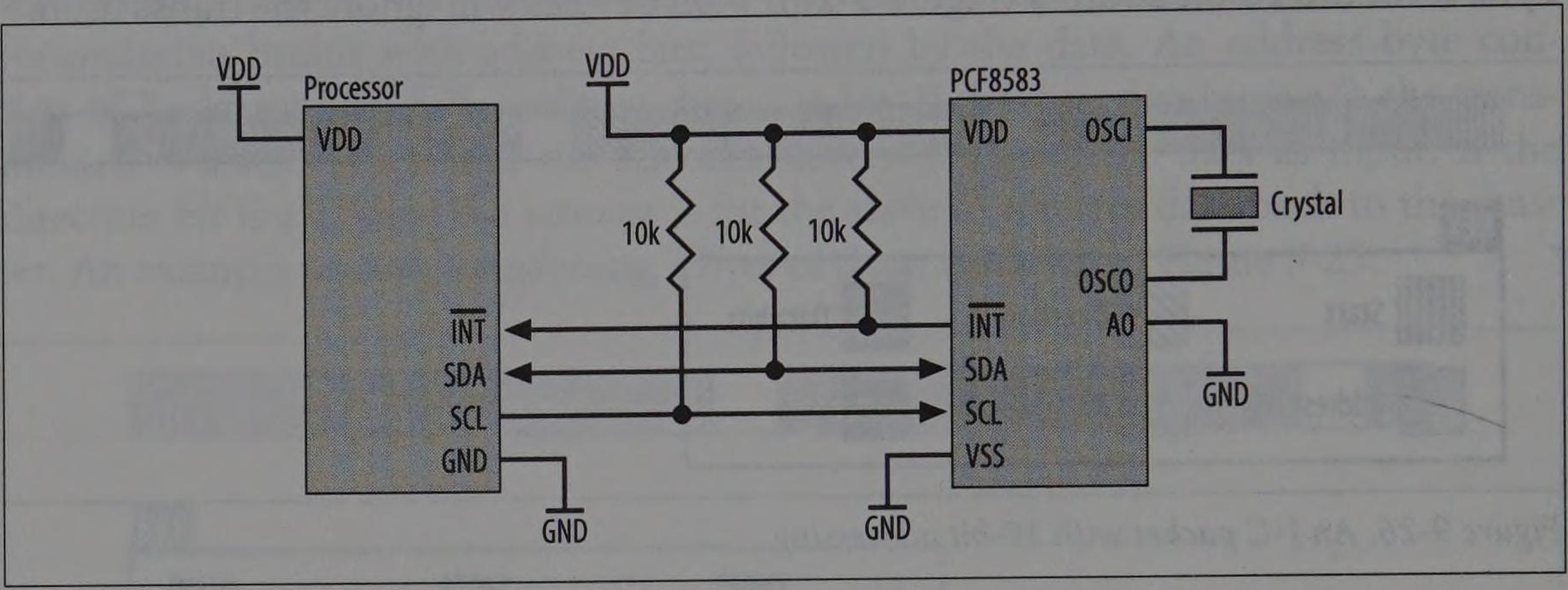


Figure 9-28. Interfacing a PCF8583 to a microcontroller

Both **SDA** and **SCL** require pull-up resistors to **VDD**. The PCF8583 also has an internal alarm function and asserts an output (**INT**) for interrupting the processor. Since this output is open-drain, a pull-up resistor is also required.

Adding a Small Display with I²C

You can use I²C to add simple LCDs (and other equivalent display technologies) to your embedded computer. These LCDs are usually just a few lines of text high but are useful for simple message display functions. Matrix Orbital (*http://www.matrixorbital.com*) produces a number of display modules that are easy to interface, such as the VFD2041. This display module is 80 characters wide by four lines deep. The interface circuit is shown in Figure 9-29, and as you can see, there's almost nothing to it. The types of LCDs found in laptops are considerably more complicated, and interfacing them to small processors is just not an option. But for simple message displays (such as on the front panel of an appliance), a circuit like this is ideal.

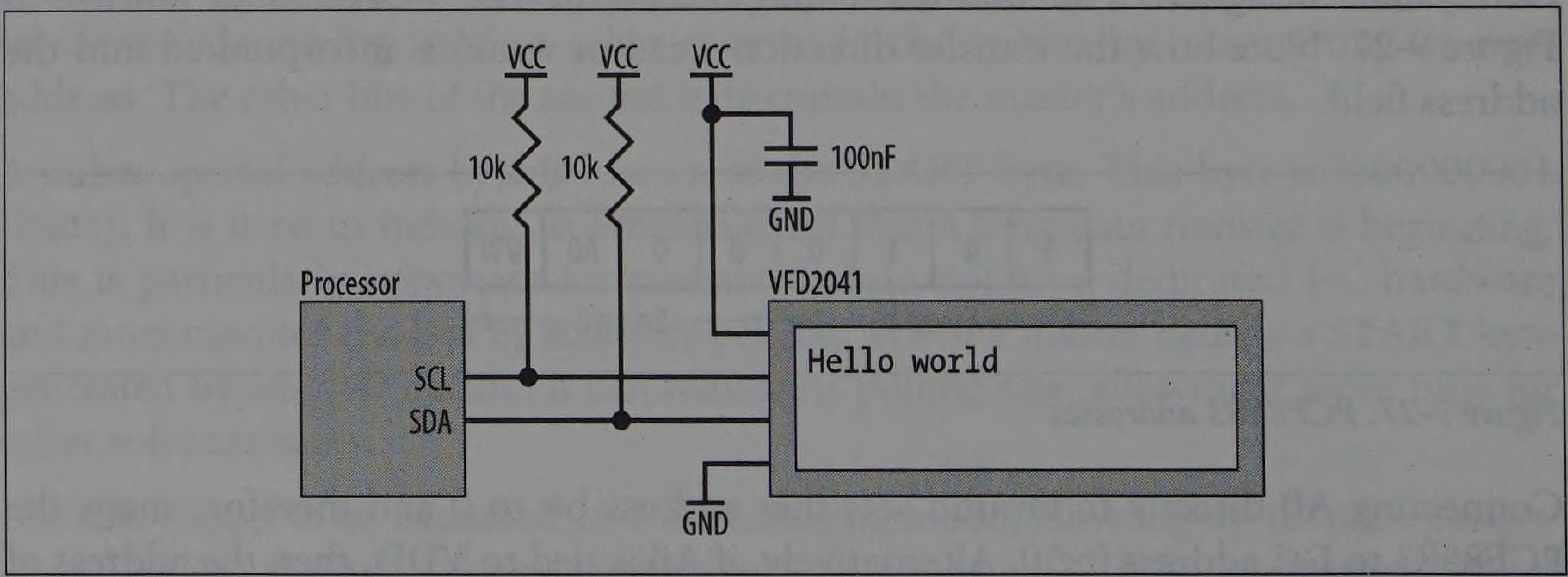


Figure 9-29. Interfacing a VFD2041 display using I²C

Many Matrix Orbital displays also come with RS-232C interfaces, so if your embedded processor doesn't support I²C, it's still easy to add a small display.

In the next chapter, well see how to add a serial port to an embedded computer and connect it to both external host computers and peripherals. We'll also look at several forms of serial interface, including RS-232C, IrDA and USB.

CHAPTER 10

Serial Ports

Yet all experience is an arch wherethro'
Gleams that untravell'd world whose margin fades
For ever and for ever when I move.
—Alfred, Lord Tennyson
Ulysses

Serial I/O involves the transfer of data over a single wire for each direction. All serial interfaces convert parallel data to a serial bit stream and vice versa. Serial communication is employed when moving data in parallel between systems is not practical, either in physical or cost terms. Such serial communication may be between a computer and a terminal or printer, the infrared beamings of a Palm computer or remote control, or, in more advanced forms, high-speed network communication such as Ethernet. For embedded computers, a simple serial interface is the easiest and cheapest way to connect to a host computer, either as part of the application or merely for debugging purposes.

This chapter looks at serial ports and how you implement an RS-232C interface. We'll even take a look at how you can power your embedded system through an RS-232C port. From there, we'll take a look at the more robust RS-422. We'll then take a look at a serial interface with a difference, IrDA. IrDA uses pulses of infrared light to transmit data across short distances, without the need of interconnecting cables. Finally, we'll take a look at a serial interface that is rapidly dominating both desktop computers and peripherals. USB allows peripherals to be networked to a host desktop computer and is becoming the standard by which you will interface your embedded computer to a Macintosh or PC.

Let's start our examination of serial interfaces by looking at the engine that drives it all.

UARTs

The simplest form of serial interface is that of the *Universal Asynchronous Receiver Transmitter*, or simply just *UART* for short. They also are sometimes called

Asynchronous Communication Interface Adapters, or *ACIAs*. They are termed "asynchronous" because no clock is transmitted with the serial data. The receiver must lock onto the data and detect individual bits without the luxury of a clock for synchronization.

Figure 10-1 shows a functional diagram of a UART. It consists of two sections, a receiver (Rx) that converts a serial bit stream to parallel data for the microprocessor and a transmitter (Tx) that converts parallel data from a microprocessor into serial form for transmission. The UART also provides status information such as whether the receiver is full (data has arrived) or the transmitter is empty (a pending transmission has completed). Many microcontrollers incorporate UARTs on-chip, but for larger systems, the UART is often a separate device.

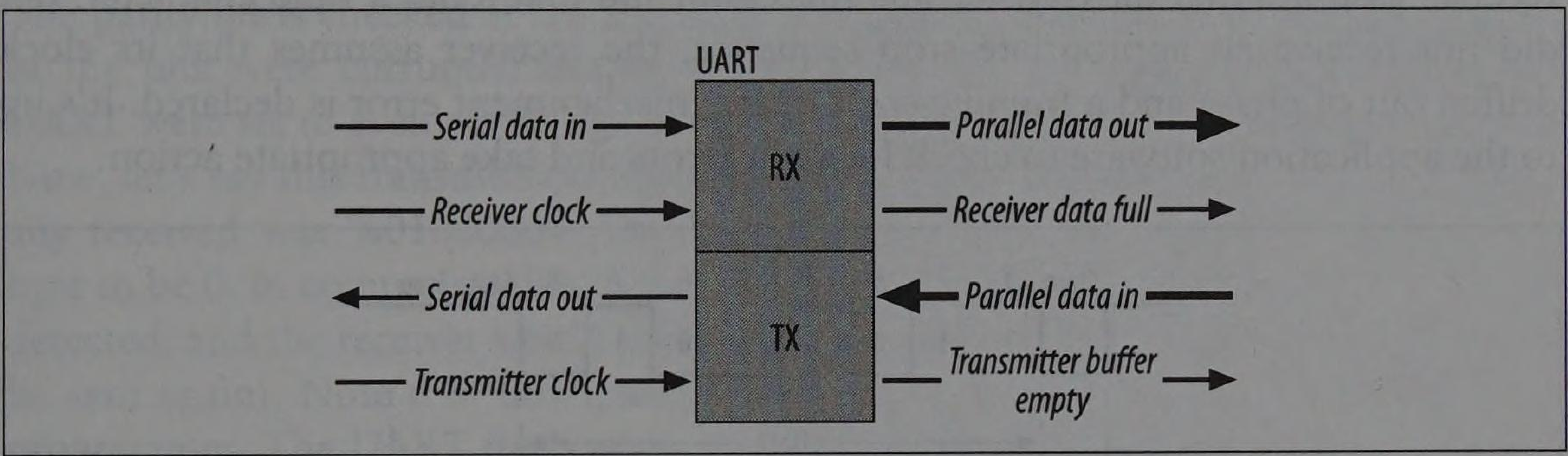


Figure 10-1. Functional diagram of a Universal Asynchronous Receiver Transmitter

Serial devices send data one bit at a time, so normal "parallel" data must first be converted to serial form before transfer. Serial transmission consists of breaking down bytes of data into single bits and shifting them out of the device one at a time. A UART's transmitter is essentially just a parallel-to-serial converter with extra features. The essence of the UART transmitter is a shift register that is loaded in parallel, and then each bit is sequentially shifted out of the device on each pulse of the serial clock. Conversely, the receiver accepts a serial bit stream into a shift register, and then this is read out in parallel by the processor.

> UARTs actually predate semiconductor-based computers. In the early days of electrical communication, UARTs were mechanical devices with cogs, relays, and electromechanical shift registers. To adjust a UART's settings, you first picked up a wrench!

One of the problems associated with serial transmission is reconstructing the data at the receiving end. Difficulties arise in detecting boundaries between bits. For instance, if the serial line is low for a given length of time, the device receiving the data must be able to identify whether the stream represents 00 or 000. It has to know where one bit stops and the next starts. The transmitting and receiving devices can accomplish this by sharing a common clock. Hence, in a synchronous serial system, the serial data stream is synchronized with a clock that is transmitted along with the

data stream. This simplifies the recovery of data but requires an extra signal line to carry the serial clock. Asynchronous serial devices, such as UARTs, do not share a common clock; rather, each device has its own, local clock. The devices must operate at exactly the same frequency, and additional logic is required to detect the phase of the transmitted data and phase-lock the receiver's clock to it.

Asynchronous transmission is used in systems in which one character is sent at a time, and the interval of time between each byte transmission may vary. The transmission format uses one start bit at the beginning and one or two stop bits at the end of each character (Figure 10-2). The receiver synchronizes its clock upon receiving the start bit and then samples the data bits (either seven or eight, depending on the system configuration). Upon receiving the stop bit(s) in the correct sequence, the receiver assumes that the transfer was successful and that it has a valid character. If it did not receive an appropriate stop sequence, the receiver assumes that its clock drifted out of phase and a *framing error* or bit-misalignment error is declared. It's up to the application software to check for such errors and take appropriate action.

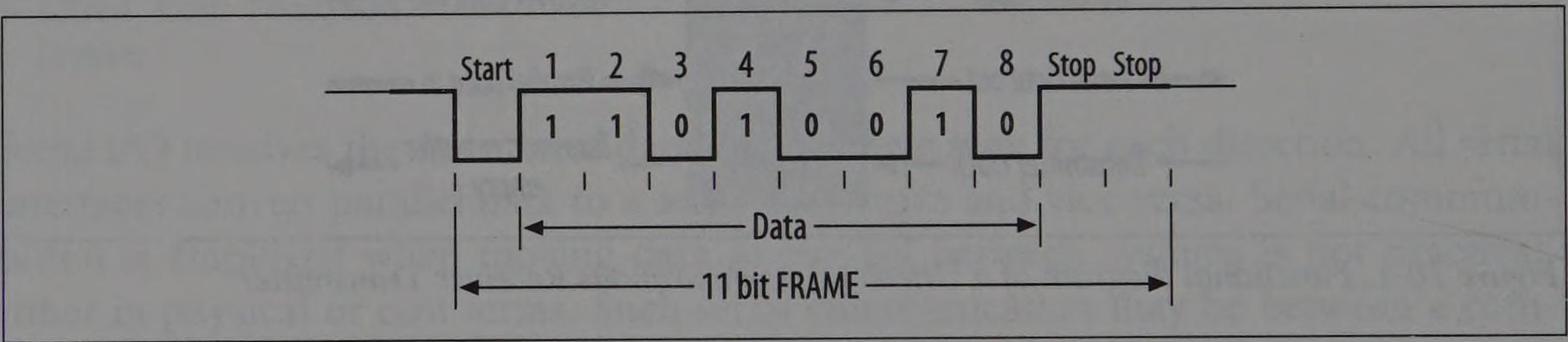


Figure 10-2. Asynchronous serial data

The conversion from parallel to serial format is usually accomplished by dedicated UART hardware, but in systems in which only parallel I/O is available, the conversion may be performed by software, toggling a single bit of a parallel I/O port acting as the serial line.

Error Detection

In any transfer of data over a potentially noisy medium (such as a serial cable), the possibility of errors exists. To detect such errors, many serial systems implement parity as a simple check for the validity of the data. The *parity* bit of a byte to be transmitted is calculated by the sending UART and included with the byte as part of the transmission. The receiving UART also calculates the parity bit for the byte and compares this against the parity bit received. If they match, the receiver assumes that everything is fine. If they do not, the receiver then knows that something went amiss and that an error exists.

There are several types of parity, the main two being *even parity* and *odd parity*. In any byte of data is either an even number of 1 bits or an odd number of 1 bits. An extra bit (the parity bit) is added to the byte to make the number of 1 bits even (even

parity) or odd (odd parity). For successful transmission, both the receiver and transmitter must be set for the same type of parity generation. There is no protocol for establishing common parity settings between UARTs; it must be done manually at either end.

So for the binary sequence %01000000, the parity bit would be 1 for even parity or 0 for odd parity. Similarly, for %11111111, the parity bit would be 0 if we were using even parity or 1 if we had odd parity. The generation and detection of parity is done automatically by dedicated hardware within the UART. It's not something you explicitly have to calculate. You do have to make sure that your UART is set to the correct type of parity generation; otherwise, it will not know how to process the parity information accordingly.

The parity bit is checked at the receiving end against the data to detect whether any of the bits were corrupted during transmission. Say we sent %01000000. If our UART were set to even parity, the calculated parity bit from %01000000 would be 1. Now, let's say this transmission was corrupted along the way, so that what was actually received was %01000001. The receiver would calculate the even parity of the byte to be 0. In comparing this to the received parity bit of 1, a parity error would be detected, and the receiver would take appropriate action (such as requesting the byte be sent again). Note that how parity errors are handled is the responsibility of the programmer. The UART itself takes no action beyond flagging the error. It is up to the software to implement appropriate error handling.

Now, what if the medium was particularly noisy and *two* bits were corrupted? Again, if we sent %01000000 with even parity (computed parity bit = 1), and this was corrupted along the way to be %01001001, the receiver would calculate the even parity of the byte to be 1. The transmission was corrupted, but *no* parity error would be detected! As you can see, the usefulness of this form of error detection is extremely limited, and for this reason more complicated error detection (and correction) schemes are often implemented. A good example of this is the *Cyclic Redundancy Check* (*CRC*) algorithm. If you need to implement CRC, there's plenty of source code available on the Web—just use your favorite search engine.

That covers the basics of how bits are transmitted serially. Now, it's time to look at how you physically implement a serial interface. We'll start with the old standard for serially connecting two computers (or just about anything else digital) together.

Old Faithful—RS-232C

RS-232C is a serial communication interface standard that has been in use, in one form or another, since the 1960s. RS-232C is used for interfacing serial devices over cable lengths of up to 25 meters and at data rates up to 38.4kbps. You can use it to connect to other computers, modems, and even old terminals (useful tools for monitoring status messages during debugging). In days of old, printers, plotters, and a

host of other devices came with RS-232C interfaces. With the need to transfer large amounts of data rapidly, RS-232C is being supplanted as a connection standard by high-speed networks, such as Ethernet. However, it can still be a useful and (importantly) simple connection tool for your embedded system.

RS-232C is *unbalanced*, meaning that the voltage level of a data bit being transmitted is referenced to local ground. A logic high for RS-232C is a signal voltage in the range -5 to -15V (typically -12V), and a logic low is between +5 and +1V (typically +12V). So, just to make that clear, an RS-232C high is a negative voltage, and a low is a positive voltage, unlike the rest of your computer's logic.

The terminology used in RS-232C also goes back to the 1960s. In those days of mainframes, a high (1) was called a "space" and a low (0) was called a "mark." You'll still find these terms kicking around in RS-232C, where you'll hear phrases like "mark parity" and "space parity." It's also not unheard of to see RS-232C systems still using 7-bit data frames (another leftover from the '60s), rather than the more common 8-bit. In fact, this is one of the reasons you'll still see email being sent on the Internet limited to a 7-bit character set, just in case the packets happen to be routed via a serial connection that supports only 7-bit transmissions. It's nice how pieces of history still linger around to haunt us! More commonly, RS-232C data transmissions use 8-bit characters, and any serial port you implement should do so too.

An RS-232C link consists of a driver and a comparator, as shown in Figure 10-3.

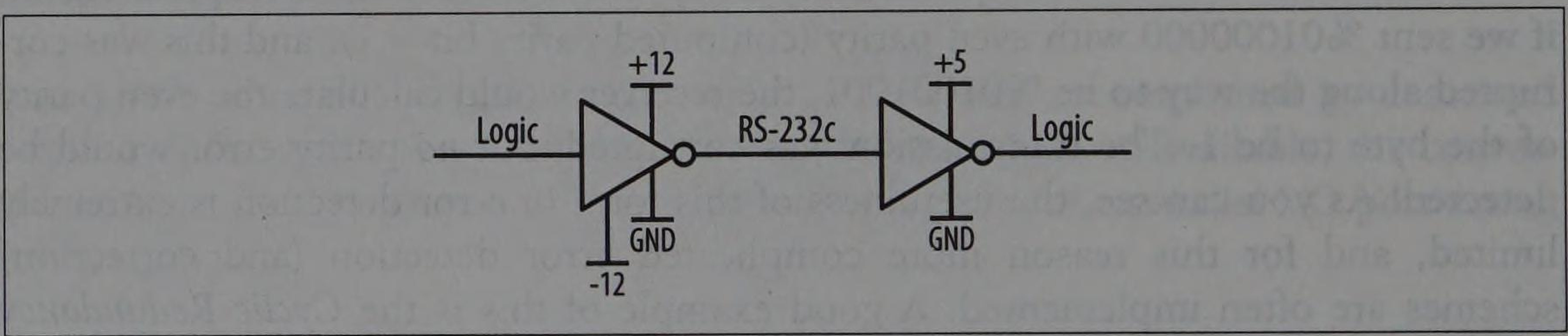


Figure 10-3. RS-232C

RS-232C also defines connectors and pin assignments, although there is a lot a room for variation (thus a lot of incompatibilities exist). RS-232C was originally intended for connecting *Data Terminal Equipment* (*DTE*) to *Data Communication Equipment* (*DCE*) (Figure 10-4). The standard therefore assumes that at one end of an RS-232C link is a DTE device and, the other, a DCE. Before the advent of computers, a DTE

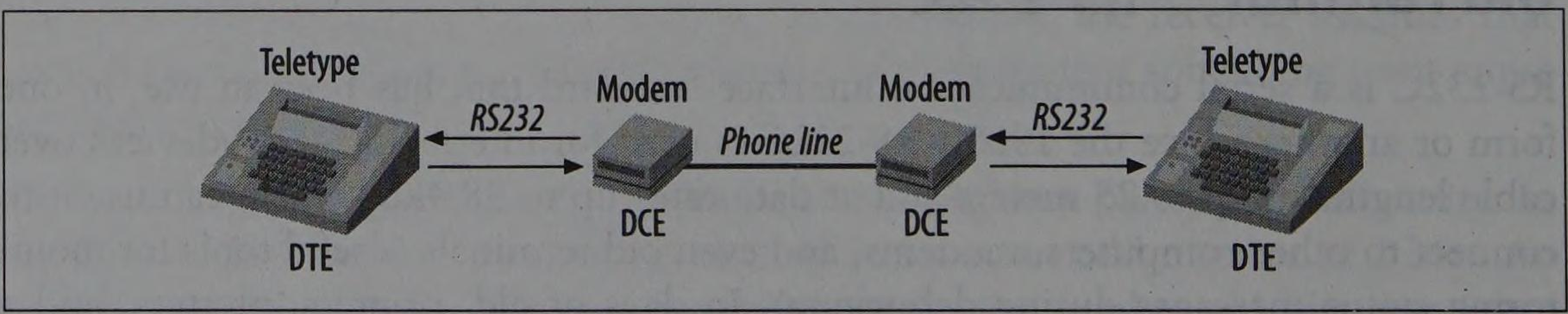


Figure 10-4. Original use of RS-232—connecting teletypes to modems

was a terminal or teletype and a DCE was a modem. The modem (MOdulator-DEModulator) provided an interface to the phone line and thereby a connection to a remote modem and terminal.

This worked simply and clearly in the days before desktop computers. The problem arises when you wish to connect either a terminal or a modem to the serial interface of a computer. Do you treat the computer as a DTE or a DCE? The RS-232C standard implies that if a terminal is at one end of the link, then the other end should be a DCE. So, if you were connecting a terminal to a Unix workstation, the RS-232C standard would like the workstation to be a DCE (Figure 10-5). Conversely, if you were connecting a modem to a computer, the computer should be a DTE (Figure 10-6). It's all a bit schizophrenic.

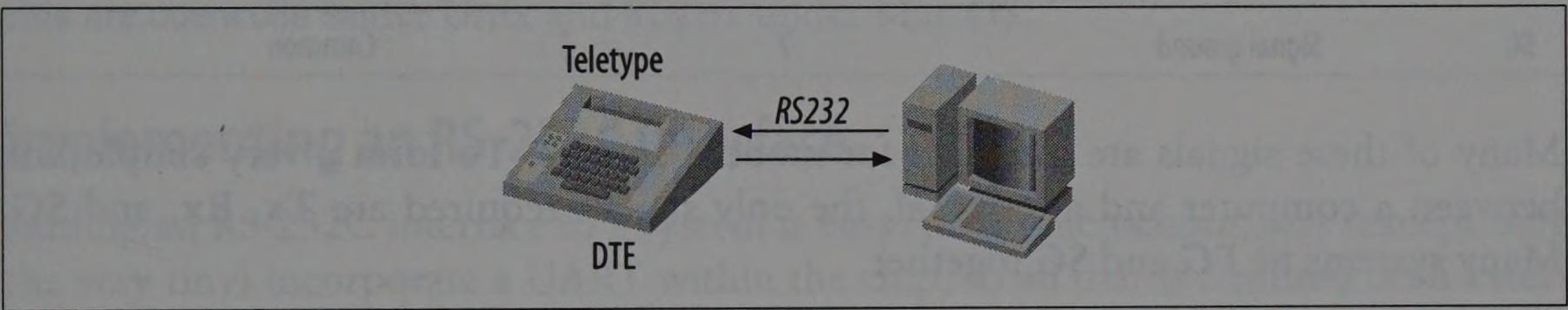


Figure 10-5. DTE device connected to a computer

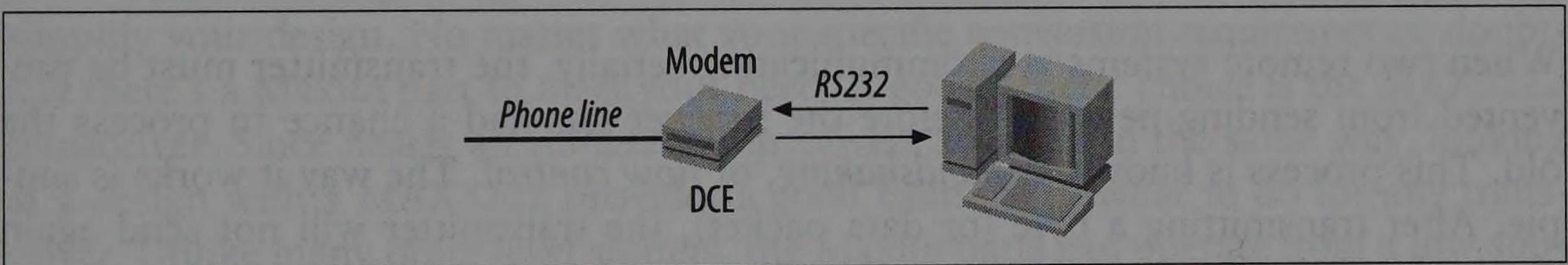


Figure 10-6. DCE device connected to a computer

Manufacturers, when faced with this problem, arbitrarily chose one or the other. The IBM PC has a DTE-type connector, whereas the makers of Unix workstations (such as Sun Microsystems) often chose to make their machines with DCE connectors, since they are more likely to be connected to terminals. To connect a PC to a modem, you need a DTE-DCE cable. To connect a PC to a terminal, you need a DTE-DTE cable. To connect a Sun workstation to a terminal, you need a DCE-DTE cable. To connect a Sun to a modem, you need a DCE-DCE cable. To connect a Sun to another Sun, you need a DCE-DCE null modem cable (where Rx and Tx cross over), and to connect a Sun to a PC, you need a DCE-DTE null modem cable. If, however, you need to connect two PCs together, you need a DTE-DTE null modem cable. So, for just two types of device (DTE and DCE), you need six types of cables to cope with the permutations! Variety, as they say, is the spice of life, but it's the bane of RS-232C!

Table 10-1 shows the "standard" connections for RS-232C, for both 25-pin and 9-pin connectors. The signal names are DTE relative. For example, **Tx** refers to data being transmitted from the DTE but received by a DCE.

Table 10-1. RS-232C signals

Signal	Function	25-pin	9-pin	Direction
Tx	Transmitted data	2	3	From DTE to DCE
Rx	Received data	3	2	To DTE from DCE
RTS	Request to send	4	7	From DTE to DCE
CTS	Clear to send	5	8	To DTE from DCE
DTR	Data terminal ready	20	4	From DTE to DCE
DSR	Data set ready	6	6	To DTE from DCE
DCD	Data carrier detect	8	1	To DTE from DCE
RI	Ring indicator	22	9	To DTE from DCE
FG	Frame ground (chassis)	1	–	Common
SG	Signal ground	7	5	Common

Many of these signals are intended for modem control. To form a very simple link between a computer and a terminal, the only signals required are **Tx**, **Rx**, and **SG**. Many systems tie **FG** and **SG** together.

Shake Hands

When two remote systems are communicating serially, the transmitter must be prevented from sending new data before the receiver has had a chance to process the old. This process is known as *handshaking*, or *flow control*. The way it works is simple. After transmitting a byte (or data packet), the transmitter will not send again until it has been given confirmation that the receiver is ready. There are three forms of handshaking: hardware, software, and none.

The no-handshaking option is obviously the most simple and is used when the transmitting system is much slower in preparing and sending data than the receiver is in processing. For example, if you had a small, embedded computer running at a pokey 1MHz and feeding data into a high-speed computer system running at 1GHz, you could assume that the faster machine would be able to keep up. However, if the faster machine is running a certain popular operating system (renowned for poor responsiveness to real-time events), it may very well not be able to keep up. In this case, handshaking would be required, and it's probably good practice to incorporate it anyway. If you're using the serial port to provide a human interface to your computer, then you can safely assume that no human will type faster than your computer can handle. So, for serial ports used solely for user access or debugging purposes, you can skip the handshaking.

Hardware handshaking in RS-232C uses two signals, *RTS* (*Request To Send*) and *CTS* (*Clear To Send*). When the transmitter wishes to send, it asserts RTS, indicating to the receiver that there is pending data. The receiver asserts CTS when it is ready, indicating to the transmitter that it may send. In this way, the flow of data is limited to the rate at which it may be processed.

Software handshaking, also known as *XON/XOFF*, is used when hardware handshaking between the transmitter and receiver is not possible, such as when the transmission occurs over a phone line. Software handshaking chooses two characters to represent a *request to suspend* transmission, and a *clear to resume*. These are normally the characters Ctrl-S (0x13) and Ctrl-Q (0x11). The caveat is that you then can't have these characters as part of the transmitted file, for they would be interpreted as flow control by the receiver and not as received data. If you're sending only ASCII text, this is not a problem, but it can be a real headache if you're sending binary data. The common solution is to preprocess the binary data prior to transmission and convert it to ASCII representation. For example, the byte 0x2F becomes the ASCII characters "2" (0x32) and "F" (0x46). Software on the receiving end converts the ASCII characters back into binary data again. Examples of software that will do this are `uuencode` under Unix and `BinHex` under Mac OS.

Implementing an RS-232C Interface

Adding an RS-232C interface to a system is easy. Most microcontrollers (except only the very tiny) incorporate a UART within the chip, so all that is required is an external level shifter to convert the serial transmissions to and from RS-232C levels. Maxim makes a huge range of RS-232C interface chips (level shifters) that greatly simplify your design. No matter what your specific conversion requirements, doubtless there's a Maxim part to meet your need. A good generic choice is the MAX3222 transceiver. Since nearly all RS-232C transceivers are used in the same way, looking at a design with a MAX3222 provides a good example of what to do for any transceiver. Unlike many other level shifters, the Maxim parts can operate from a low supply voltage, in the range 3.0V to 5.5V. Many other manufacturers' devices need supplies of +12V and -12V and therefore require additional voltage regulators. The MAX3222 consumes minimal power (1mA in normal operation and as low as 1μA in shutdown mode), making it ideal for portable and battery-powered applications. If you do not need to shut down the serial port into low-power operation, the MAX3232 can be substituted. It is functionally the same, except that it lacks shutdown capability.

Using the MAX3222 is trivial, as there is almost no design work involved at all. The only external support components required are capacitors for the chip's internal charge pumps. These pumps generate the +12V and -12V voltages required for RS-232C transmission, without requiring (additional) external voltage regulators. Figure 10-7 shows the schematic.

The capacitor C1 must be a minimum of 0.1μF. If operating the chip at less than 3.6V, C2, C3, and C4 can also be 0.1μF. If the supply voltage is to be as high as 5.5V, the C2, C3, and C4 must be a minimum of 0.47μF. Since these are minimum values, larger capacitors may be used. However, if C1 is increased, then the remaining capacitors must also be increased accordingly. C5, the decoupling capacitor for

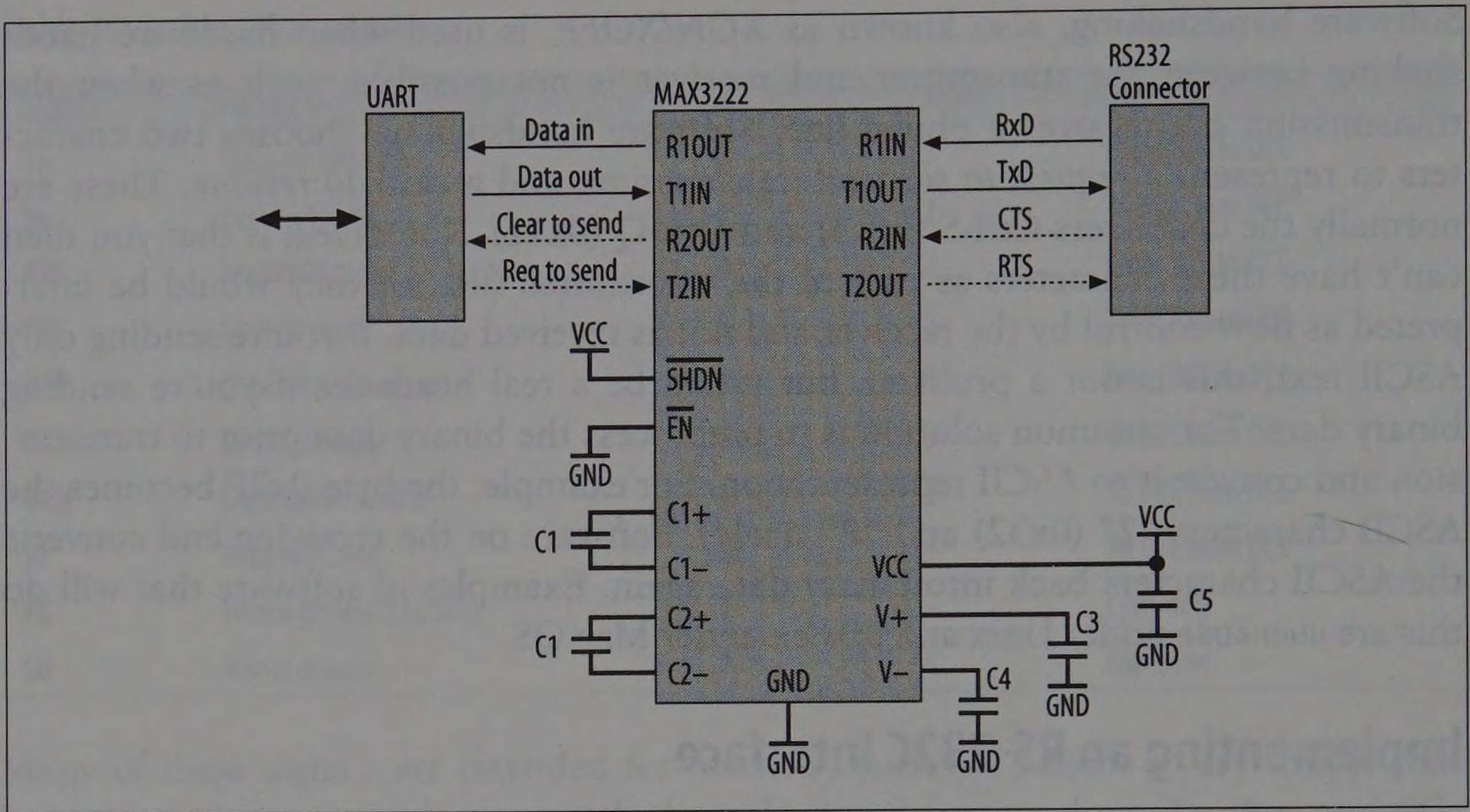


Figure 10-7. RS-232C interface using a MAX3222

VCC, is nominally 0.1μF. All capacitors should be as close to the appropriate pins of the chip as possible.

The only remaining connections are the serial data lines from the UART and the signals to the RS-232C connector. If implementing a minimal serial interface, only **Rx**, **Tx**, and ground are required. **RTS** (Request to Send) and **CTS** (Clear to Send) are optional. The RS-232C connector may be either a 25-pin or a 9-pin DB connector (it looks like the letter *D* in shape). However, the connector could also be just a row of pins, a parallel header, or even just wires soldered directly onto the PCB.

The MAX3222 has two control inputs, **$\overline{\text{SHDN}}$** (shutdown) and **EN** (enable). **$\overline{\text{SHDN}}$** places the RS-232C transmitters in high impedance, thereby disabling them. This reduces the chip's current consumption to less than 1μA. When in shutdown mode, the receivers are still active. Thus, the UART is still able to receive data even if the MAX3222 is in low-power mode. If **$\overline{\text{SHDN}}$** is not required, just connect it directly to **VCC**.

Similarly, **EN** is used to control the receiver outputs. Placing **EN** high puts the receiver outputs into high impedance, while the transmitter outputs are unaffected. To enable the receivers, **EN** is asserted (pulled low). If disabling the receivers is not required, then tie **$\overline{\text{EN}}$** to ground to permanently activate them.

If needed, **$\overline{\text{SHDN}}$** and **EN** may be controlled by a microcontroller's I/O lines or by simple digital outputs using a latch.

The MAX3222 is sufficient to implement a minimal RS-232C interface, using just **Rx**, **Tx**, and ground. It also has additional drivers to support **RTS** and **CTS**, allowing for basic flow control. Should you require a full RS-232C interface, the MAX3241 is a good choice. Its operation is similar to the MAX3222, but it has additional

transceivers allowing the inclusion of **DTR**, **DSR**, **DCD**, and **RI** for modem control. The MAX3421 may also be used to interface to a serial mouse, since it is able to meet the appropriate voltage and current requirements.

Using a Serial Port as a Power Supply

If an embedded system is to be permanently connected to a host computer via an RS-232C serial interface, the embedded system may be parasitically powered from the serial interface. Many RS-232C signals go unused and can supply a moderate amount of current (nominally 50 mA, but it can vary and, as always, you should check the specific device to which you are interfacing). If your embedded system requires less than this for its total current draw, you can use an RS-232C control signal for power.

For instance, the ***RTS*** (*Request To Send*) or ***DTR*** (*Data Terminal Request*) signals may not be used in many RS-232C applications. Either can be used as the power input to a voltage regulator and thereby provide the system with power. The host computer therefore uses **RTS** of its serial port as the power control for the embedded system. Under software, the host sends **RTS** high, and the embedded system is powered up. Send **RTS** low, and the embedded system is switched off. The catch to all this is to ensure that your embedded system's current draw is low enough so that it can be powered by **RTS**. The advantage of this technique is that you require no external power supply for your embedded system. It works, as if by magic, whenever plugged into a serial port. The other catch is that you can't then use that RS-232C control signal for its original purpose. It must turn on and stay on to provide your computer with power.

The schematic for this is shown in Figure 10-8, which also includes an RS-232C interface for a microcontroller, using a MAX3232. Note the diode, D1. Since **RTS** will be a negative voltage (as low as -15V) when low, some protection is required for the voltage regulator, since it is not designed to have its input taken below zero volts. The diode can be any garden-variety power diode, such as a 1N4004, and will conduct only when **RTS** is positive. The voltage regulator (MAX604) converts the voltage from **RTS** to a supply of 3.3V for the embedded system. If we required a supply of 5V, we'd simply use a MAX603 instead. The circuit would otherwise be the same. The output of the regulator is smoothed by the capacitor C5, and a power-on LED is provided to show us when we have power. The MAX3232 sits between the RS-232C port and the processor, level-shifting the serial transmissions from the processor's logic levels to RS-232C and vice versa.

There we have the basics of RS-232C. It's a very common interface that is easy to use, but it does have its limitations and quirks. It was originally intended for connecting dumb terminals and teletypes to modems, not for interconnecting computers and peripherals. A better choice is RS-422, designed for more robust and versatile serial connections.

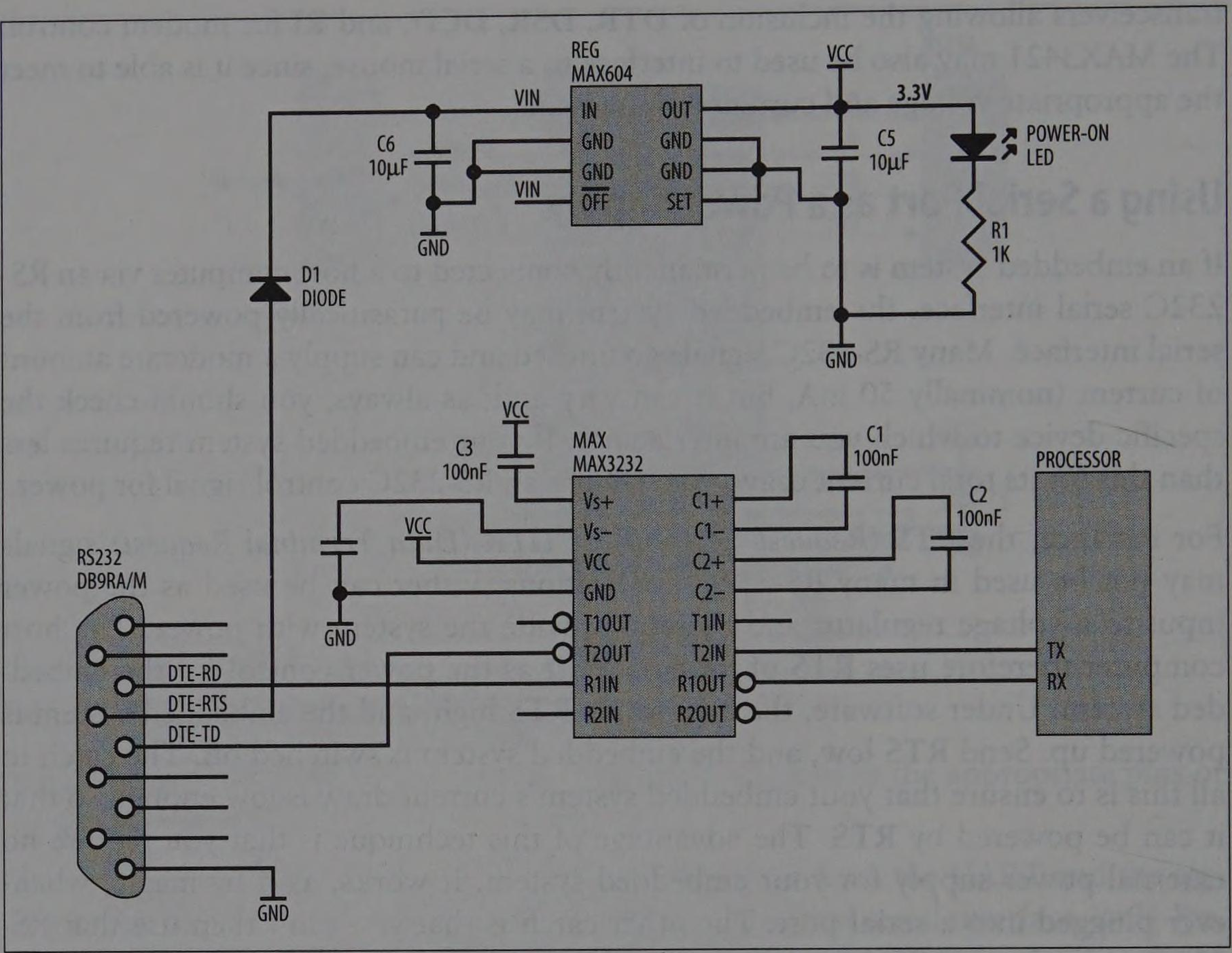


Figure 10-8. Using RTS as a power source in a low-powered embedded system

RS-422

Unlike RS-232C, which is referenced to local ground, RS-422 uses the difference between two lines, known as a *twisted pair* or a *differential pair*, to represent the logic level. Thus, RS-422 is a *balanced transmission*, or in other words, it is not referenced to local ground. Any noise or interference will affect both wires of the twisted pair, but the difference between them will be less affected. This is known as *common-mode rejection*. RS-422 can therefore carry data over longer distances and at higher rates with greater noise immunity than RS-232C. RS-422 can support data transmission over cable lengths of up to 1200 meters (approximately 4000 feet).

Figure 10-9 shows a basic RS-422 link, where a driver (D) is connected to a receiver (R) via a twisted pair. The resistor, R_t, at the receiving end of the twisted pair is a termination resistor. It acts to remove signal reflections that may occur during transmission over long distances and is required. R_t is nominally 100–120Ω.

The voltage difference between an RS-422 twisted pair is between ±4V and ±12V between the transmission lines (Figure 10-10). RS-422 is, to a degree, compatible with RS-232C. By connecting the negative side of the twisted pair to ground, RS-422 effectively becomes an unbalanced transmission. It may then be mated with RS-232C.

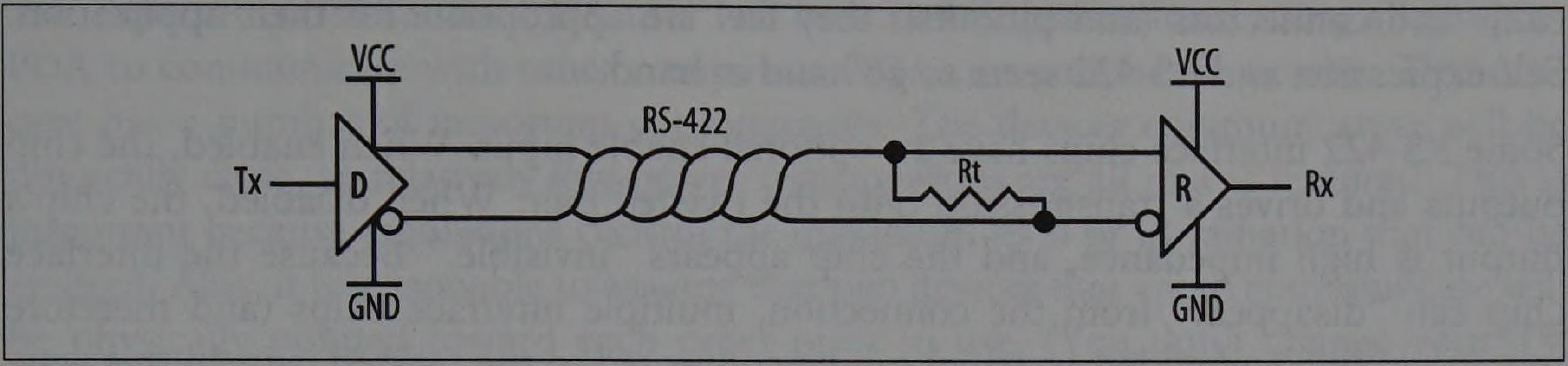


Figure 10-9. RS-422

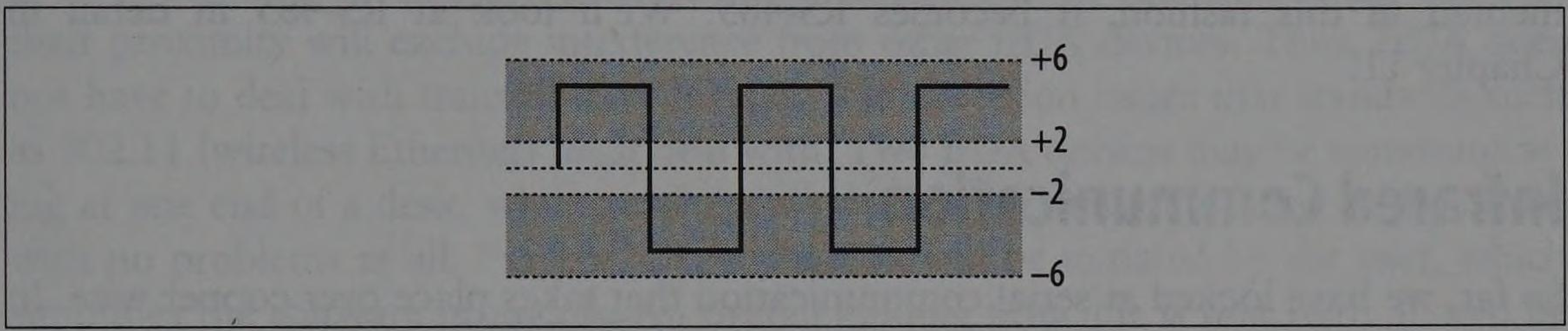


Figure 10-10. RS-422 voltage levels

Since the voltage levels of RS-422 fall within the acceptable ranges for RS-232C, the two standards may be interconnected. RS-422 was the serial interface found on Apple Macintosh computers but was quietly dropped with the coming of the iMacs.

A wide variety of RS-422 interface chips is available. Figure 10-11 shows a simple RS-422 bidirectional interface implemented using two Maxim MAX3488s. The **Tx** and **Rx** pairs of each MAX3488 are connected to UARTs within each embedded system, just as we did with RS-232C.

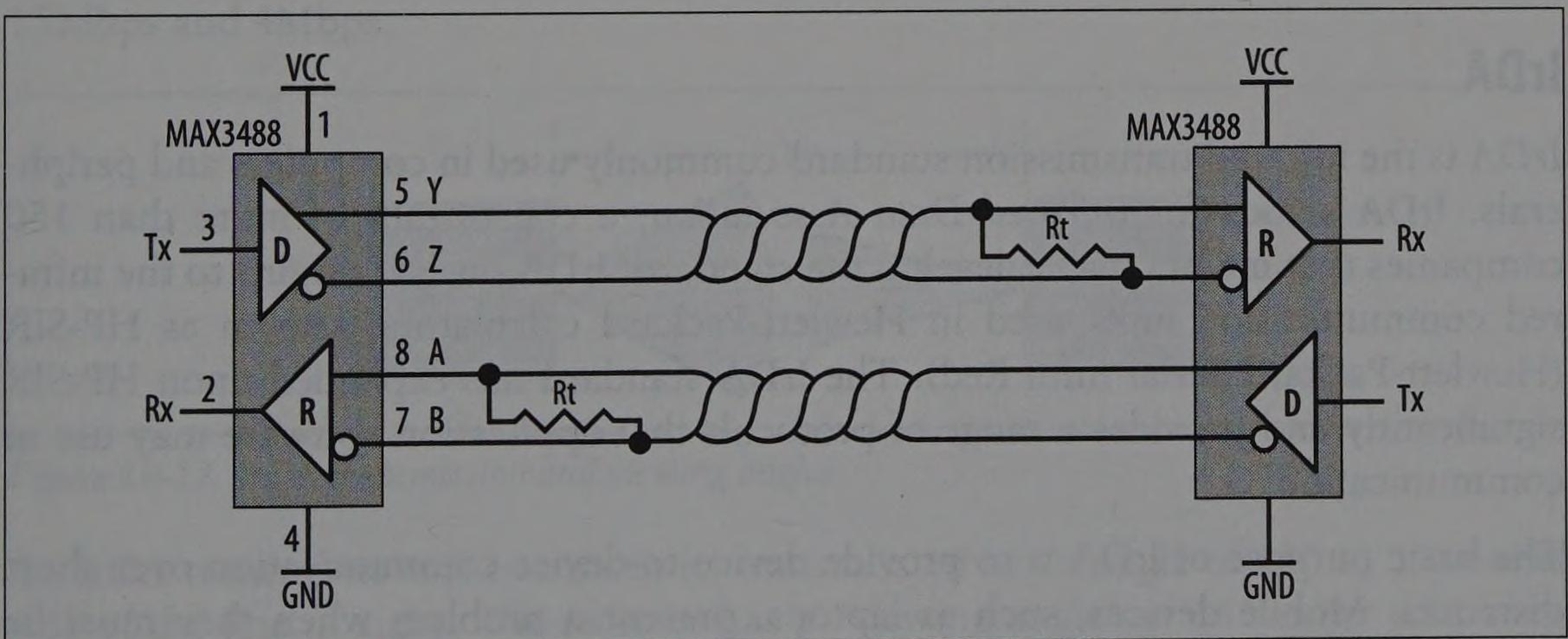


Figure 10-11. Bidirectional RS-422 interface

An important note: RS-422 specifies only the voltages for the standard, not the physical, implementation (pinouts or connectors). That is covered by RS-449. Now, no one seems to bother with RS-449, mainly because it is unnecessarily complex for most uses. People using RS-422 just seem to do their own thing, picking whatever

cable and connectors (and pinouts!) they feel are appropriate for their application. Self-expression and RS-422 seem to go hand in hand.

Some RS-422 interface chips have an optional enable input. When enabled, the chip outputs and drives a transmission onto the twisted pair. When disabled, the chip's output is high impedance, and the chip appears "invisible." Because the interface chip can "disappear" from the connection, multiple interface chips (and therefore more than two embedded systems) can be connected to the twisted pair. In this way, RS-422 can be extended into a low-cost, robust, simple network. When implemented in this fashion, it becomes RS-485. We'll look at RS-485 in detail in Chapter 11.

Infrared Communication

So far, we have looked at serial communication that takes place over copper wire. In this section, we'll look at serial communication using infrared light. Infrared (IR) transmission of data is becoming commonplace, and IR transceivers are appearing in laptop computers, PDAs, and cell phones. They are also appearing in peripherals such as printers and network interfaces, allowing no-fuss/no-cable connection for people on the move. IR communication is also used by remote controls to talk to their appliances. Your TV, VCR, and DVD remotes all have an IR LED to beam commands across the room.

We'll start our discussion of IR communication by looking at the most common standard. Later, we'll see just how trivial infrared hardware is to implement.

IrDA

IrDA is the infrared transmission standard commonly used in computers and peripherals. IrDA stands for *Infrared Data Association*, a consortium of more than 150 companies that maintain and develop the standard. IrDA owes it origins to the infrared communication links used in Hewlett-Packard calculators, known as HP-SIR (Hewlett-Packard Serial Infra Red). The IrDA standard has expanded upon HP-SIR significantly and provides a range of protocols that application software may use in communication.

The basic purpose of IrDA is to provide device-to-device communication over short distances. Mobile devices, such as laptops, present a problem when they must be connected to other machines or networks. Chances are, the correct cable is not at hand, or one of the machines is not configured correctly to allow networking. When the users are nontechnical types, this can be a real problem. IrDA was developed as the solution. With IrDA, no cables are required, and standard protocols ensure that devices can exchange information seamlessly. Full details of the IrDA standard and protocols are available from *http://www.irda.org*.

The expectation is that the IrDA user will be a mobile professional using a laptop or PDA to communicate with other computers, PDAs, or peripherals nearby. This concept has a number of important consequences. The devices communicating will be physically close, so relatively low-power transmissions are all that is required. This is important because regulations control the maximum level of IR radiation that can be emitted. Also, it is reasonable to assume that two devices that are to communicate will be physically pointed toward each other prior to use. (You don't change your TV channel by aiming the remote at the cat, unless, of course, your cat is especially reflective.) It can also be assumed that only two devices will be communicating and that their proximity will exclude interference from other IrDA devices. Thus, IrDA does not have to deal with transmission collision and detection issues that standards such as 802.11 (wireless Ethernet) must deal with. Two IrDA devices may be communicating at one end of a desk, while another two devices are communicating at the other, with no problems at all. Further, a transmission will be initiated by the user, which simplifies the software protocols. An overall guiding principle is that IrDA should be cheap to implement, since it must find its way into low-cost consumer devices.

With all that in mind, IrDA is a point-to-point protocol that uses asynchronous serial transmission over short distances. The initial IrDA specification (1.0) supported data rates of between 2400bps and 115.2kbps over distances of 1 meter, although some IrDA transceivers can achieve greater distances than this. Initial IR communication takes place at 9600bps, and devices negotiate the data rate up or down, depending on their capabilities and needs. Unlike RS-232C, the user does not need to set, know about, or even care what bit rate is being used in communication. Since its original specification, the standard has been expanded to support higher data rates of 1. 15Mbps and 4Mbps.

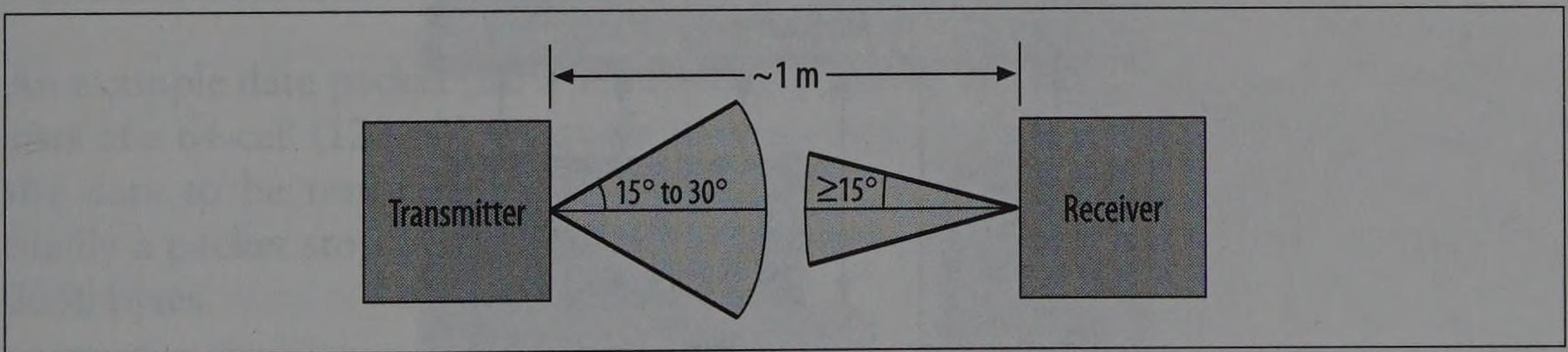


Figure 10-12. IrDA transmission and viewing angles

An IrDA transmitter will beam out its transmission at an angle of 15° to 30° either side of the line of sight. The receiver has a "viewing angle" of 15° either side of its line of sight (Figure 10-12). So, if two IrDA devices are placed a meter or less apart and generally aimed in each other's direction, communication will not be a problem.

The IrDA standard specifies a number of protocol layers for communication. The IrPHY (IR Physical Layer) specification details the hardware layer, including requirements for modulating the outputs of UARTs prior to transmission. The control protocol is known as High-level Data Link Control, or HDLC. IrLAP (Infrared Link

Access Protocol) uses HDLC for controlling access to the communication medium. One IrLAP exists per device. An IrLAP connection is essentially master/slave configuration, or as they are known in IrDA parlance, *primary* and *secondary devices*. The primary device starts communication, sends commands, and handles data-flow control (handshaking). It is rare for a primary device to be anything other than a computer. Secondaries (such as printers) simply respond to requests from primaries. Two primary-type devices can communicate by one primary assuming the role of a secondary device. Typically, the device that initiates the transfer remains the primary, while the other device becomes a secondary for the duration of the transaction.

IrLMP (Infrared Link Management Protocol) provides the device's software with a means of sharing the single IrLAP between multiple tasks that wish to communicate using IrDA. IrLMP also provides a query protocol by which one device may interrogate another to determine what services are available on the remote system. This query protocol is known as LM-IAS, or Link Management Information Access Service. These are the basic IrDA protocols that all devices must support. Beyond these, IrDA also provides a number of optional services. IrCOMM provides emulation of standard serial port and parallel port devices. For application software, the IR port can then be used as if it were just another serial or parallel port. Using IrCOMM, a laptop or PDA can communicate with an IR-enabled printer just as though that printer was physically plugged into the mobile computer. IrLAN allows access to local area networks via the IR interface. IrOBEX provides a mechanism for object exchange between devices, in software that supports object-oriented programming. Finally, TinyTP is a lightweight protocol allowing applications to perform flow-control (handshaking) when transferring data from one device to another. Figure 10-13 shows how these protocol layers fit together.

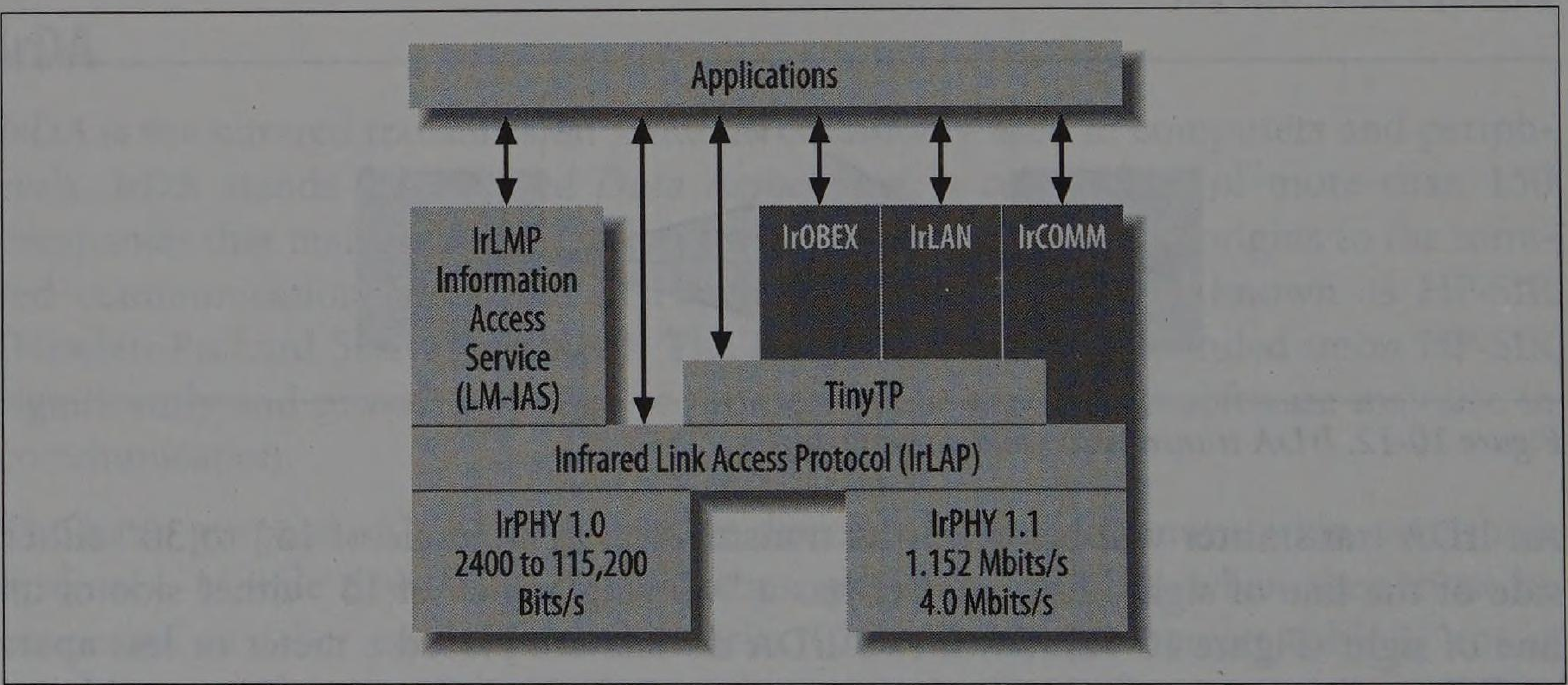


Figure 10-13. IrDA protocol layers

At the lower data rates, all protocol handling, packet forming, and error checking is done in software by the processor within an IrDA-compliant device. At higher data rates, dedicated hardware performs these functions, since low-cost embedded

processors may not have the computing horsepower to complete these tasks in the time available.

Since IrDA communicates using light, there must be some way to distinguish between a logic 0 and a logic 1 during transmission. To solve this problem, IrDA uses a bit-encoding scheme known as *Return-to-Zero*, or *RZ*. With RZ, a frame consists of a transmission interval that is divided into subintervals representing individual bits. A logic 0 is represented by a pulse that is 3/16 the width of a bit sub-interval, while a logic 1 is represented by the absence of a pulse (Figure 10-14).

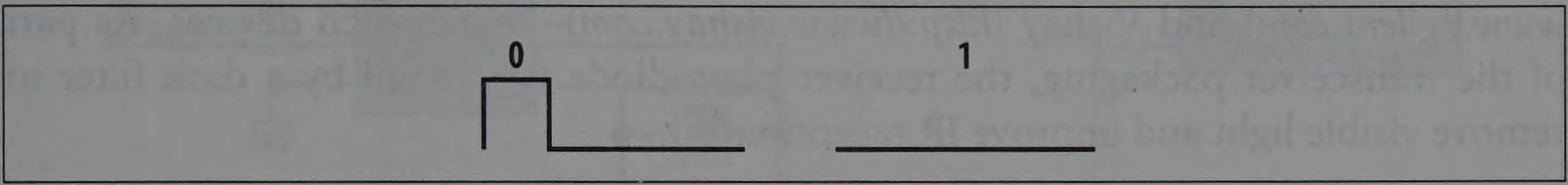


Figure 10-14. RZ encoding

At data rates of 4Mbps, *PPM*, or *Pulse Position Modulation*, is used to distinguish different bits. With PPM, the position of the pulse is varied. Its location within the sub-interval determines the transmitted bit pattern. The PPM used in IrDA is known as *4PPM* and uses one of four positions to provide the transmission of two data bits (Figure 10-15). In PPM terminology, these are known as *cells*.

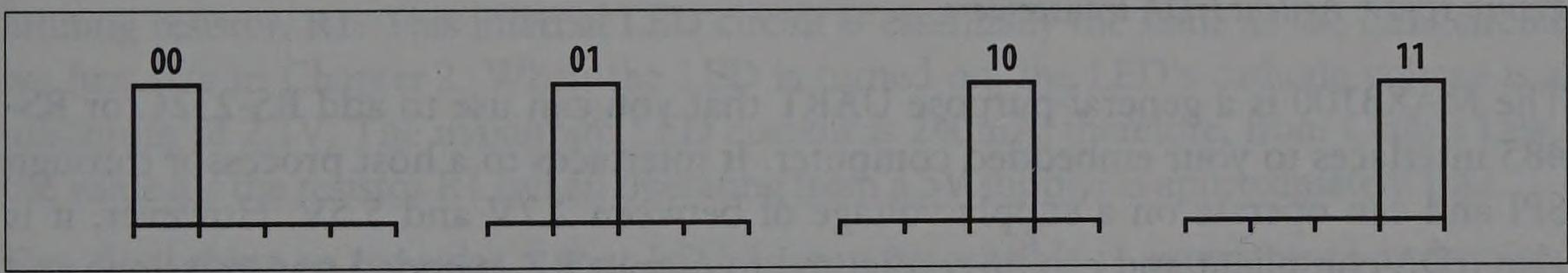


Figure 10-15. 4PPM cell encoding

An example data packet (for a 4Mbps transmission) is shown in Figure 10-16. It consists of a 64-cell (128-bit) preamble packet, a start packet, the frame body containing the data to be transmitted, a 32-bit Cyclic Redundancy Check (CRC) code, and finally a packet stop marker. The data frame can be as little as 2 bytes or as large as 2050 bytes.

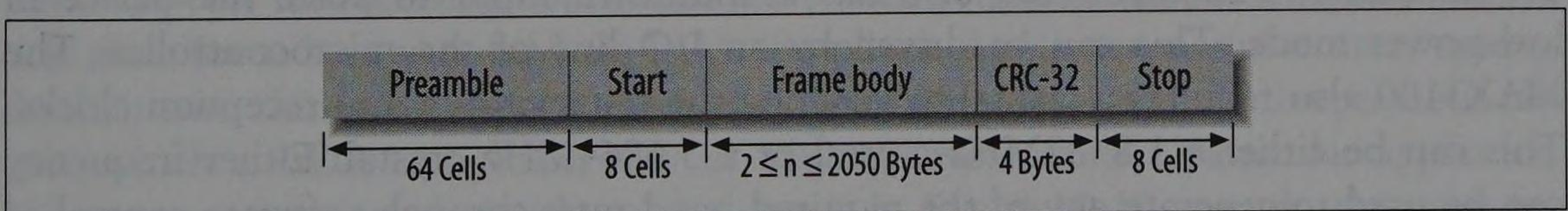


Figure 10-16. A 4Mbps data packet

Now, most UARTs are not capable of performing transmissions in RZ or 4PPM encoding. Therefore, a special device, known as an *EnDec* (*Encoder-Decoder*), converts the standard UART output to RZ and vice versa. A good EnDec to choose is the HSDL-7001 from Agilent or the MCP2120 by Microchip. Some UARTs, such as the

MAX3100, incorporate an EnDec on-chip and so may be used to directly interface to an IR transceiver.

An IrDA Interface

For IR transmission and reception, you can use an individual IR LED and an IR photodiode detector. Alternatively, you can use combined IR transceivers that incorporate both the IR LED and photodiode, along with support components, in a convenient package (Figure 10-17). Several manufacturers—including Agilent (*http://www.agilent.com*) and Vishay (*http://www.vishay.com*)—make such devices. As part of the transceiver packaging, the receiver photodiode is covered by a dark filter to remove visible light and improve IR reception.

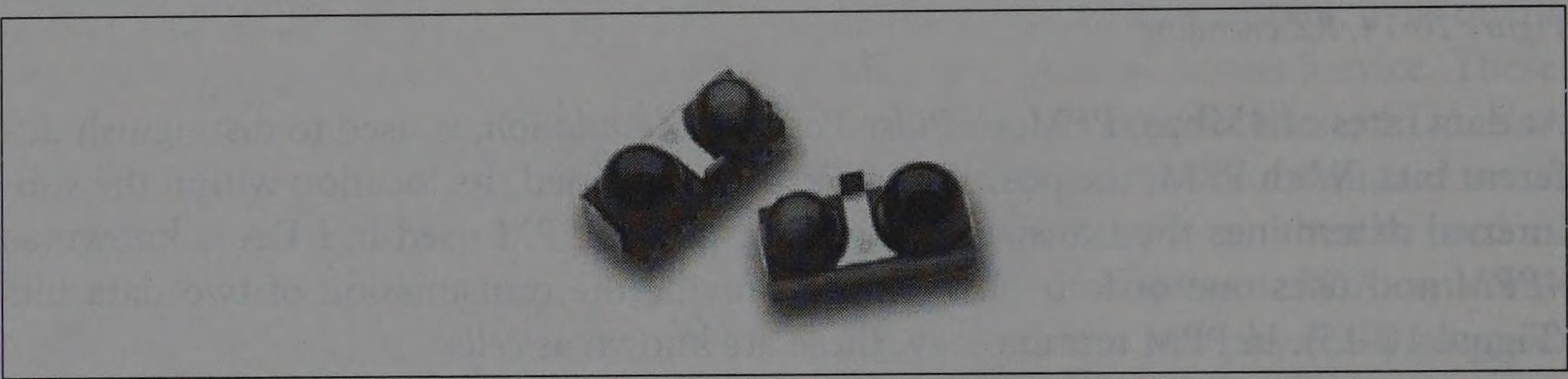


Figure 10-17. Agilent IrDA transceivers

The MAX3100 is a general-purpose UART that you can use to add RS-232C or RS-485 interfaces to your embedded computer. It interfaces to a host processor through SPI and can operate on a supply voltage of between 2.7V and 5.5V. However, it is also IrDA compliant and can be configured to output RZ-encoded transmissions and receive RZ-encoded bit streams. All you need to do to make an IrDA interface is add an IR transceiver, some inverter gates, and a few support components. The schematic for the circuit is shown in Figure 10-18.

On the left of the schematic are standard SPI connections to a microcontroller. The MAX3100 also has an interrupt output by which it can notify the host processor of a change in state (such as it has received data). This interrupt line is pulled high by a 10kΩ resistor. The MAX3100 also has a shutdown input to place the device in low-power mode. This can be driven by an I/O line of the microcontroller. The MAX3100 also requires a crystal to generate the transmission and reception clocks. This can be either a 1.8432MHz crystal or a 3.6864MHz crystal. Either frequency can be used to generate any of the required baud rates through software control of the internal clock dividers. The lower-speed crystal will cause the MAX3100 to use less power.

A number of IR transceivers are available, and in this schematic I have chosen to use the Agilent HSDL-1001. To interface the MAX3100 to the HSDL-1001, we simply need to invert both the transmit (**TX**) and receive (**RX**) signals. The HSDL-1001 has a shutdown input that is used to put the receiving photodiode in low-power mode. It has no effect on

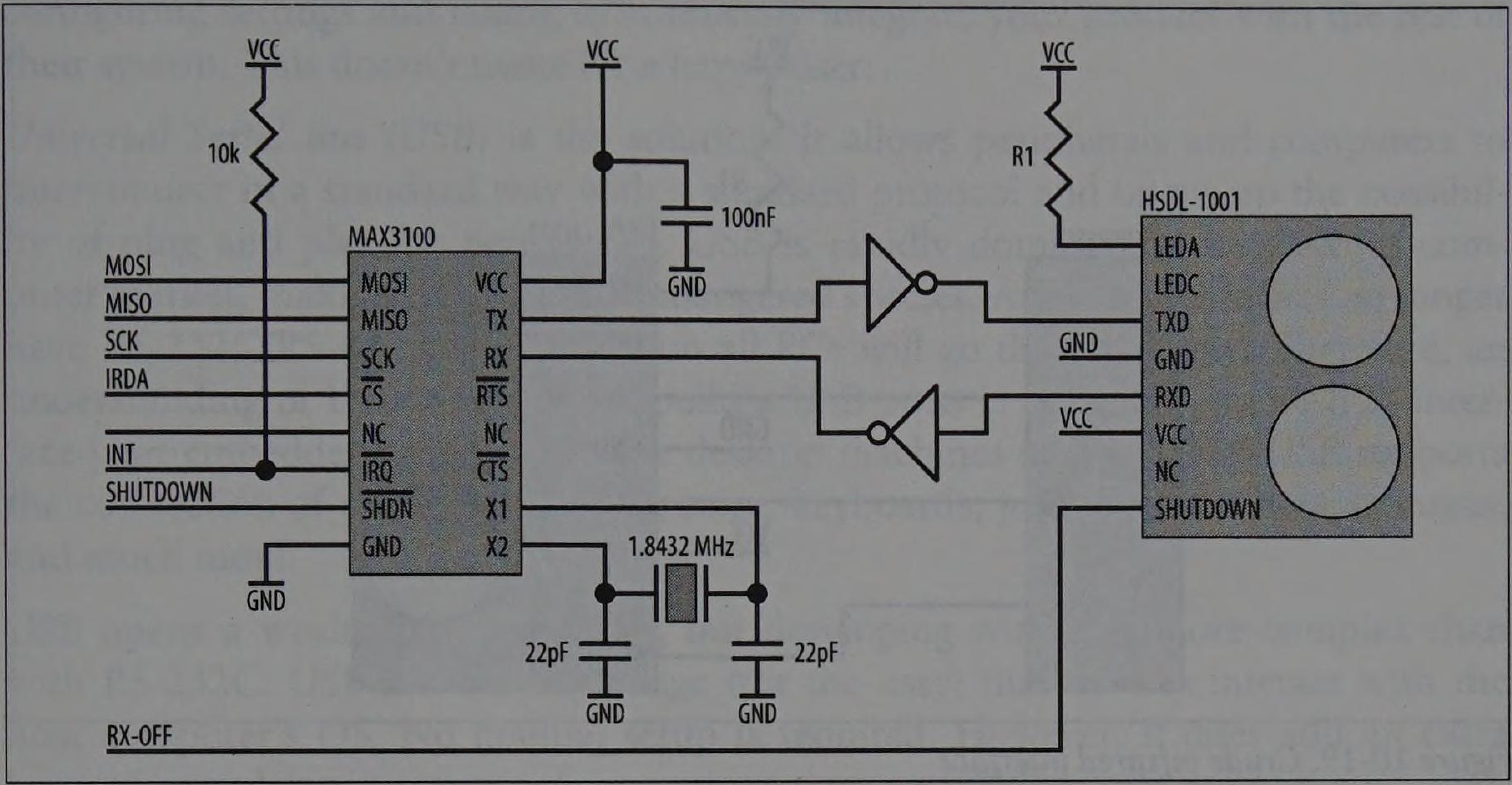


Figure 10-18. IrDA interface for an embedded computer

the transmitting LED, however. This shutdown input may also be driven by an I/O line from the processor. For maximum versatility, this shutdown is independently controlled to the MAX3100's shutdown. The transmitter LED of the HSDL-1001 requires a current-limiting resistor, R1. This internal LED circuit is essentially the same as the LED circuit we first saw in Chapter 2. When the LED is turned on, the LED's cathode voltage is a minimum of 2.1V. The maximum LED current is 240mA; therefore, from Ohm's Law, the value for the resistor R1 (when operating from a 5V supply) is approximately 15Ω.

One final thing to consider is that IrDA is very susceptible to interference and noise; therefore, all power supplies should be properly decoupled using capacitors for every power pin. Ground planes should also be used to shield the transceiver and associated signal tracks.

Other Infrared Devices

Your TV, VCR, DVD player, and air conditioner and a host of other devices all have infrared ports for receiving commands from their remote controls. The bad news is that none (or at least very few) are IrDA compliant. Appliance manufacturers tend to do their own thing and often at their own weird baud rates too. So the previous circuit, which is IrDA compliant, may or may not work with a particular appliance. However, something as simple as the circuit in Figure 10-19 may do the trick for you.

The transmitter of the HSDL-1001 may be driven directly by a processor I/O line. Similarly, the receiver may be sampled using an I/O line (as an input) too. So, under software control (and by "manually" toggling the transmitter I/O line as appropriate), the HSDL-1001 may be fed the correct RZ bit stream at the appropriate bit rate. This manual technique is commonly used in standard serial interfaces to implement a serial port on processors that don't have a UART (and that can't be expanded

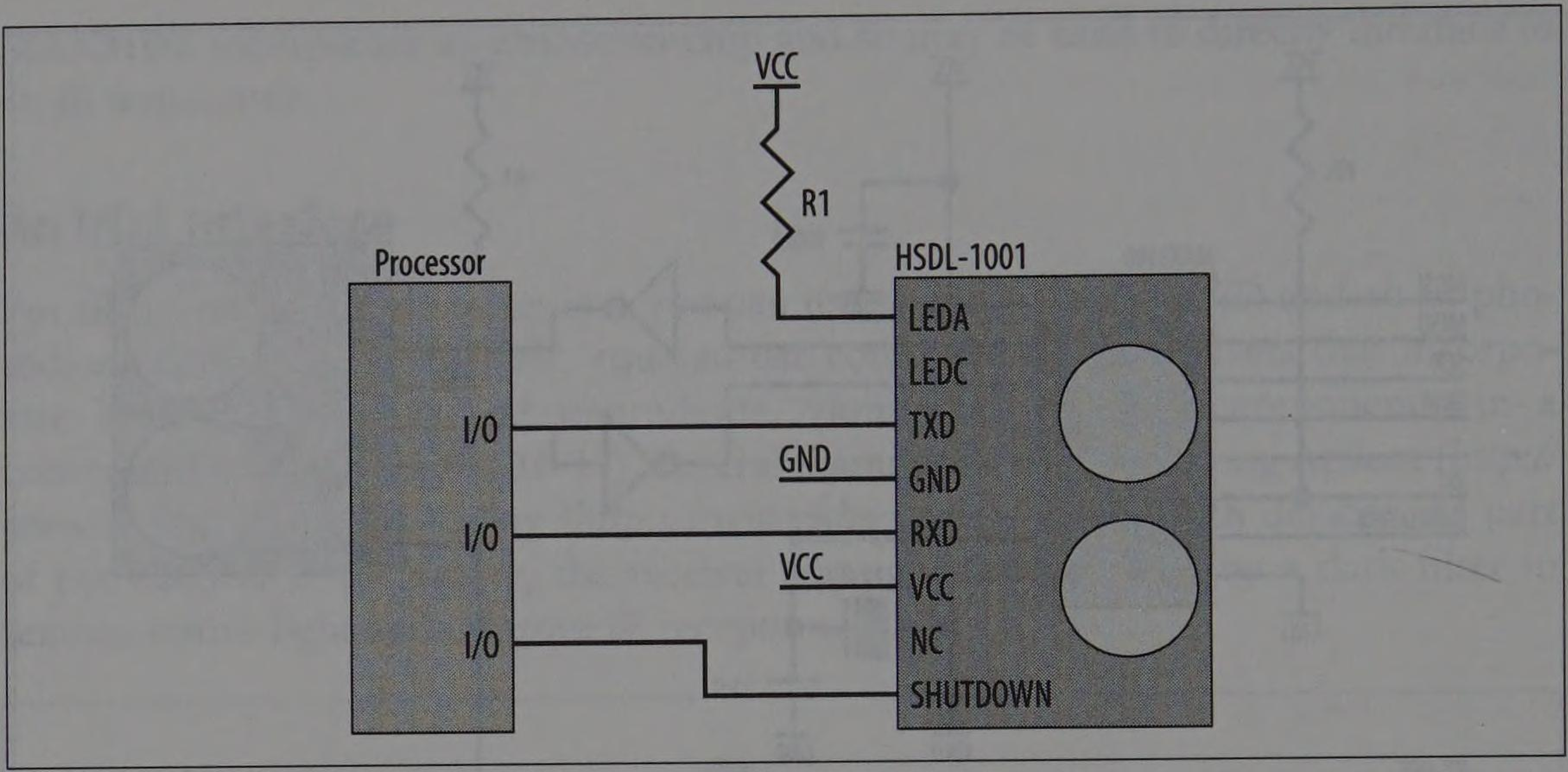


Figure 10-19. Crude infrared interface

upon). This software UART technique can just as easily be extended to an infrared interface. It's up to you as the programmer to ensure that you get the timing correct.

> You can't see the IR output of a remote control with the naked eye, but if you point a camcorder at it, you can. Point the remote into the camera lens and hit a button or two. If you look in the viewfinder while doing this, you can clearly see the control beaming its bits. The way this works is that the CCD imager inside the camera is sensitive to IR as well as visible light. That's one of the reasons camcorders are able to shoot so well in low-light levels. To further increase their ability to image at night, some camcorders have IR lights on the front to illuminate the darkness, yet be invisible to people looking on.
>
> Try the trick with the remote. You'll be surprised at just how bright the IR LEDs really are.

For information on appliance (and remote control) IR protocols and programming, go to *http://www.remotecentral.com*.

USB

At the start of this chapter, we looked at RS-232C, that old standard of communication that's not so standard after all. RS-232C has lots of problems and lots of limitations. Getting any two RS-232C devices to talk is not easy. You need the right cable with the right sort of connectors, and then you need to manually coordinate the communication parameters such as data rate, parity, and handshaking. At best it is a nuisance, at worst, a headache. For hardware manufacturers, it presents a dilemma. Your goal in developing your product should be to make that product as easy to use as possible. You don't want users stumbling around with incorrect cables, manually

configuring settings and failing to seamlessly integrate your product with the rest of their system. This doesn't make for a happy user.

Universal Serial Bus (*USB*) is the solution. It allows peripherals and computers to interconnect in a standard way with a standard protocol and opens up the possibility of plug and play for peripherals. USB is rapidly dominating the desktop computer market, making RS-232C an endangered species. Apple Macintoshes no longer have RS-232C/RS-422 ports, and soon all PCs will go the same way. Therefore, an understanding of USB (and how to build a USB port) is critical if you wish to interface your embedded computer to the desktop machines of the future. USB supports the connection of printers, modems, mice, keyboards, joysticks, scanners, cameras, and much more.

USB opens a wealth of possibilities, but developing with it is more complex than with RS-232C. USB has the advantage (for the user) that devices interact with the host computer's OS. No manual setup is required. However, it does add an extra layer of complexity to your software, since your embedded code must interact with the host in the appropriate way. USB can even provide power to peripherals through the same cable as data. No external power supply (or power cable) is required. So, for the user, a USB peripheral is simplicity itself.

In this section, we'll just take an overview of USB and then go on to see how you incorporate a USB interface into your embedded system. The protocols and specifications for USB are long and complex and well beyond the scope of this book. Fortunately, to design USB-based hardware, the task is much simpler. We'll simply take an overview and then look at a physical USB implementation. For a full look at the standard, a list of vendors, and more documentation than you can shake a cable at, visit *http://www.usb.org*.

There are two specifications for USB: USB 1.1 and USB 2.0. USB 2.0 is fully compatible with USB 1.1. USB supports data rates of 12Mbps and 1.5Mbps (for slower peripherals) for USB 1.1 and data rates of 480Mbps for USB 2.0. Data transfers can be either isochronous* or asynchronous.

USB is a high-speed bus that allows up to 127 devices to be connected (Figure 10-20). No longer is having only one or two ports on your computer a limitation. Further, one standard for cables and connectors eliminates the confusion that existed with RS-232C. Devices are able to self-identify to a host computer and can be *hot-swapped*, meaning that the systems do not need to be powered down before connection or disconnection.

The basic structure of a USB network is a tiered star. A USB system consists of one or more USB devices (peripherals), one or more hubs, and a host (controlling computer). The host computer is sometimes known as the *host controller*. Only one host may exist

* Meaning occurring at equal intervals of time.

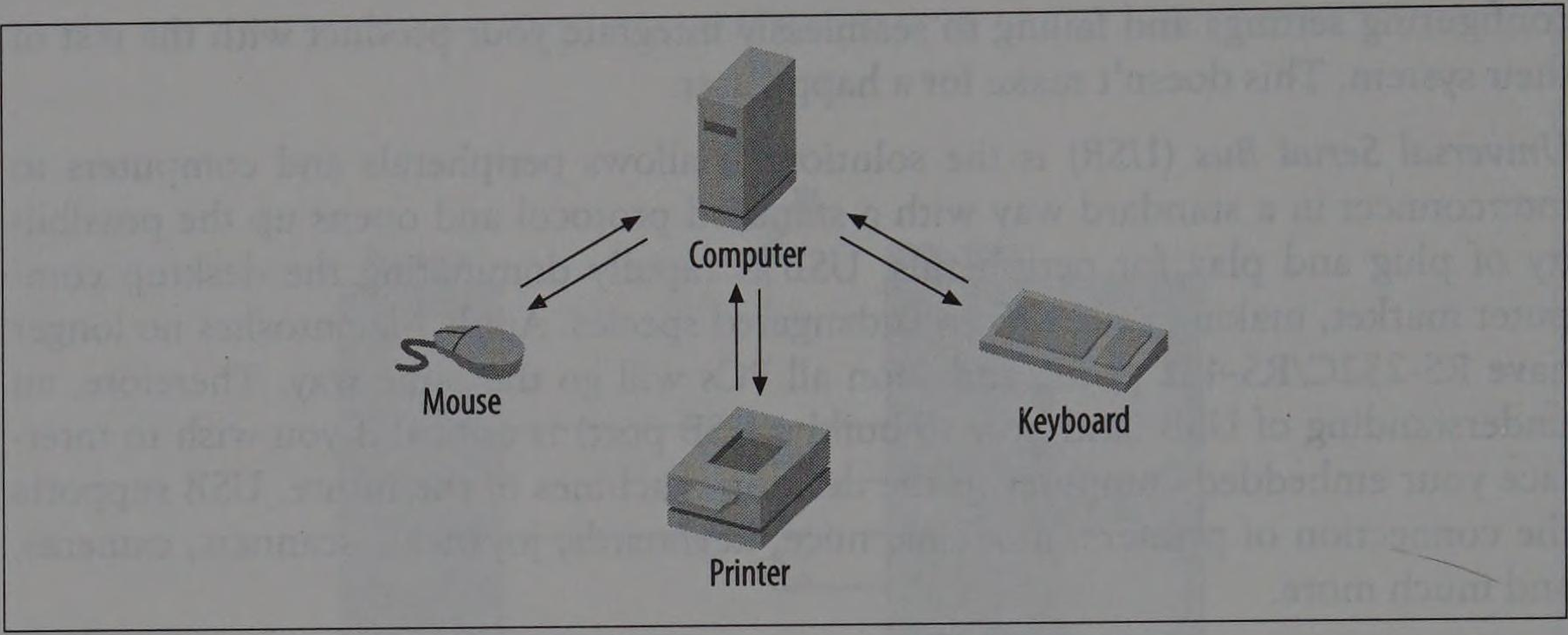


Figure 10-20. USB allows a host to connect with a variety of peripherals

in a USB network. The host controller incorporates a *root hub*, which provides the initial attachment points to the host. The hubs form nodes into which devices or other hubs connect and are (largely) invisible to USB communication. In other words, traffic between a device and a host is not affected by the presence of hubs.

Hubs are used to expand a USB network. For example, a given host computer may have five USB ports. By connecting hubs, each with additional ports, to the host's ports, the physical connectivity of the system is increased (Figure 10-21). Many USB devices (such as keyboards), incorporate built-in hubs, allowing them to provide additional expansion as well as their primary function.

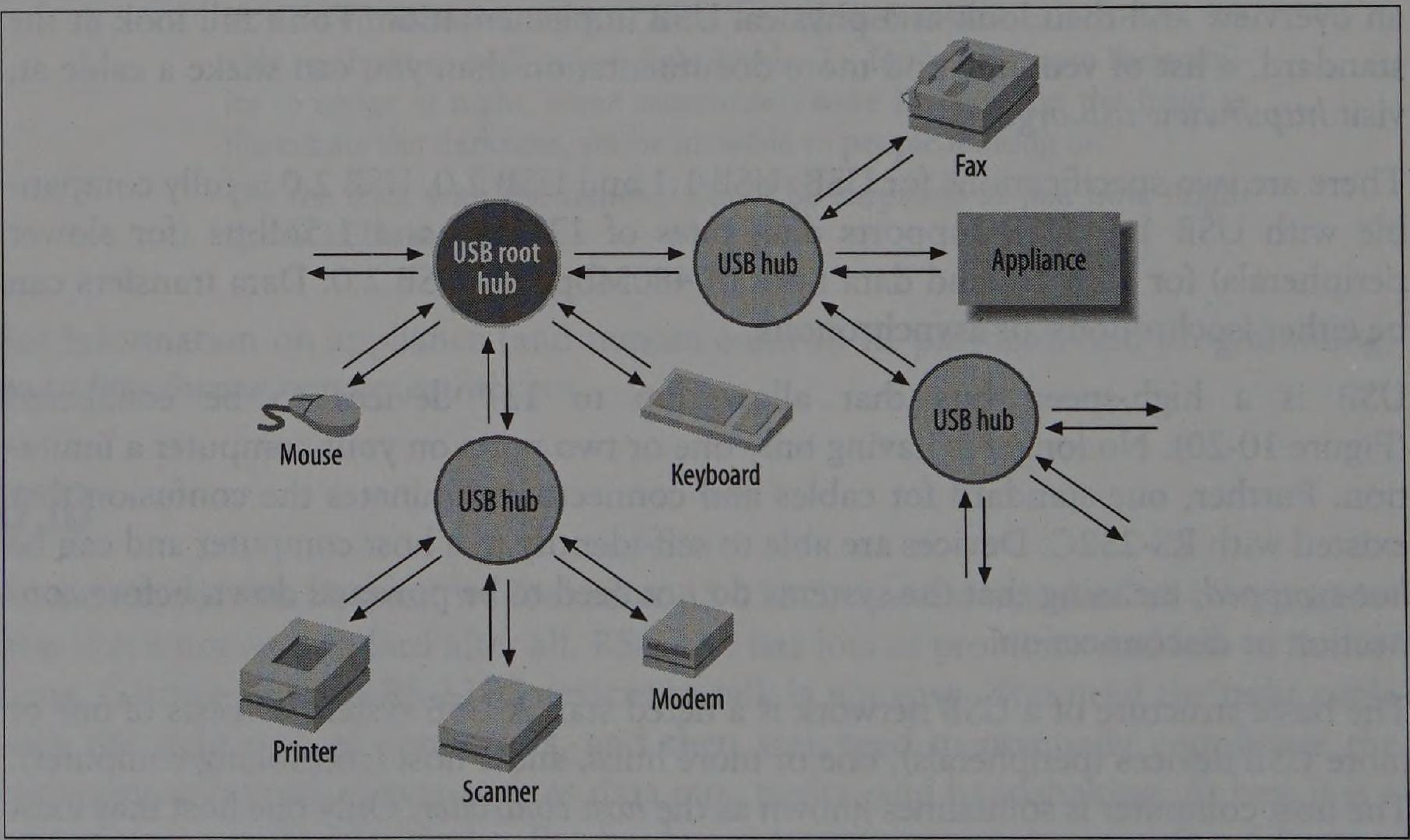


Figure 10-21. USB is expandable using hubs

The host will regularly poll hubs for their status. When a new device is plugged in to a hub, the hub advises the host of its change in state. The host issues a command to enable and reset that port. The device attached to that port responds, and the host retrieves information about the device. Based on that information, the host operating system determines what software driver to use for that device. The device is then assigned a unique address, and its internal configuration is requested by the host. When a device is unplugged, the hub advises the host of the change in state when polled, and the host removes the device from its list of available resources. The detection and identification of USB devices by a host is known as *bus enumeration*.

USB "knows" about and supports different classes of devices. Each class represents the functionality that the device can provide to the host. Some example classes (and example devices) are listed in Table 10-2. A single, physical, USB peripheral can encompass several classes.

Table 10-2. USB device classes

Class	Purpose
Audio	Audio and music devices, sound systems
Chip/Smart Card Interface Devices (CCID)	Smart card devices
Common Class (CCS)	Generic devices
Communications Device	Modems, telephones, and network interfaces
HID	Human Interface Devices (HIDs) such as mice and keyboards
Hub	USB hub
IrDA	Infrared devices
Mass Storage	Hard disks, CD-ROMs, DVD-ROMs
Monitor	Computer monitors and display devices
Physical Interface Devices	Joysticks and other devices (such as motion platforms) that provide physical feedback
POS Terminals	Point-of-Sale (POS) devices such as cash registers and EFTPOS devices
Power	Devices with power control or monitoring (battery backup and recharging, for example)
Printer Class	Printers
Imaging Class	Scanners and cameras

USB Packets

Four types of transfers can take place over USB. A *control transfer* is used to configure the bus and devices on the bus and to return status information. A *bulk transfer* moves data asynchronously over USB. An *isochronous transfer* is used for moving time-critical data, such as audio data destined for an output device. Unlike a bulk transfer, which can be bidirectional, an isochronous transfer is unidirectional and

includes no cyclic redundancy check (CRC) field. An *interrupt transfer* is used to retrieve data at regular intervals, ranging from 1 to 255 milliseconds.

Data is transferred between USB devices using packets. A transfer can consist of one or more packets. A packet consists of a SYNC (synchronization) byte, a PID (Packet ID), content (data, address, etc.), and a CRC.

The SYNC byte phase locks the receiver's clock. This is equivalent to the start bit of an RS-232C frame. The PID indicates the function of the packet, such as whether it is a data packet or a setup packet, for example. The upper 4 bits of the packet ID are the inverse of the lower 4 bits, for additional error checking. For example, the packet ID for a data packet is 0x3C. In binary, this is %0011 1100.

USB packets can be one of four types: token, data, handshaking, and preamble.

Tokens are 24-bit packets that determine the type of transfer that is to take place over the bus. There are four types of token packet (Figure 10-22). A token packet consists of a SYNC byte, a packet ID (indicating packet type), the address of the device being accessed by the host, the end-point address, and a 5-bit CRC field. The end-point address is the internal destination of the data within the device.

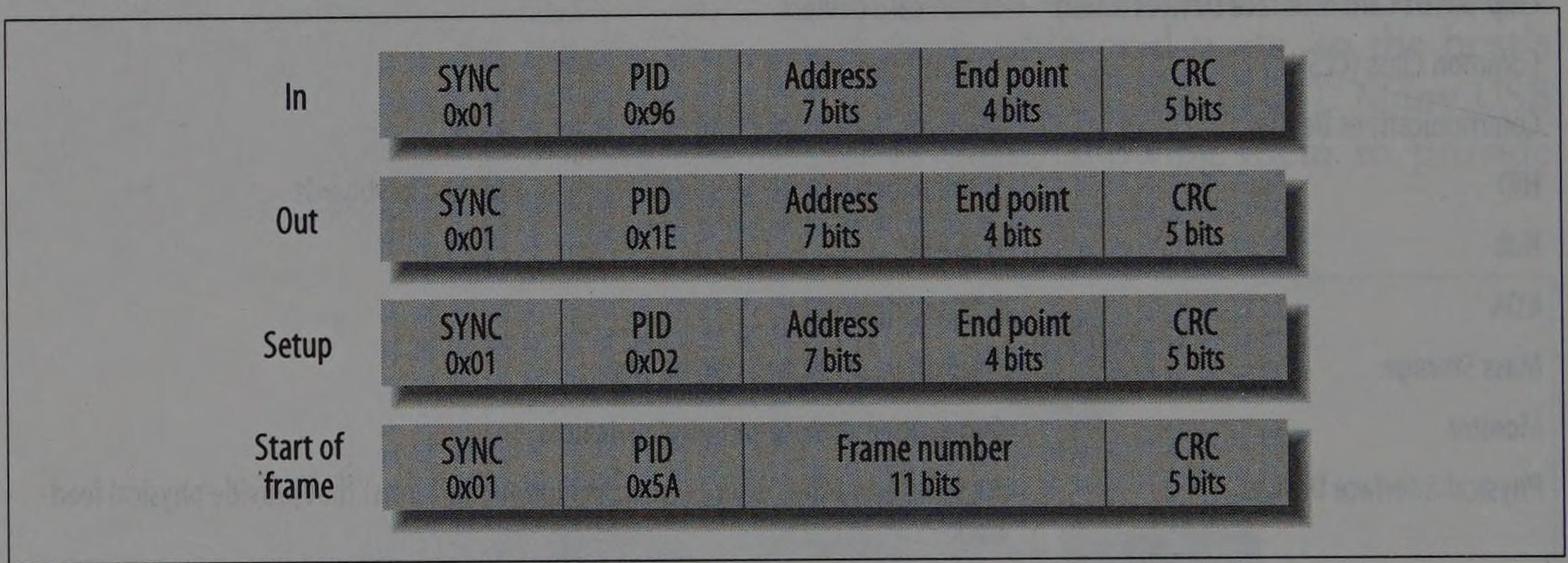


Figure 10-22. USB token packets

There are two types of *data packet*, known as DATA0 and DATA1 (Figure 10-23). The transmission of data packets alternates between the two types. A single data packet can transfer between 0 and 1023 bytes. The data packet CRC is 16 bits, due to the larger packet size.

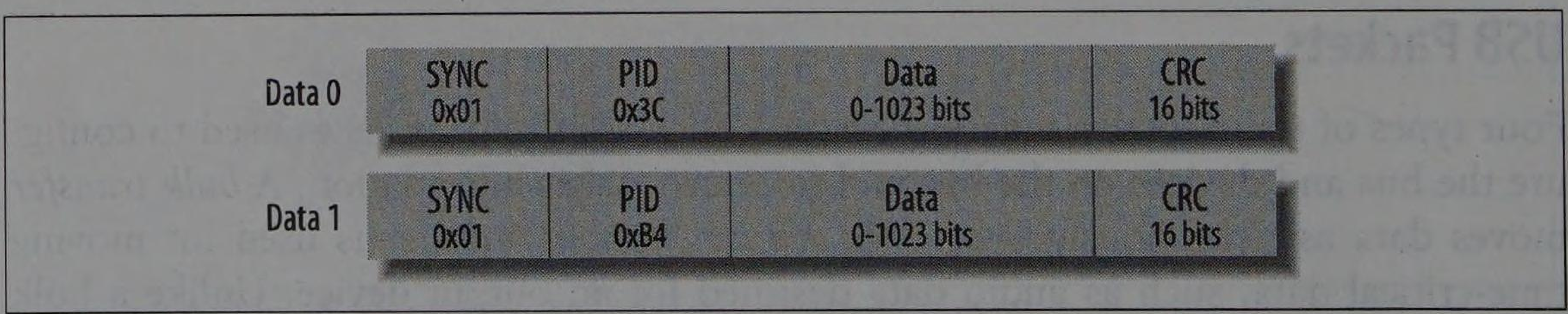


Figure 10-23. USB data packets

There are three types of *handshaking packet* (Figure 10-24). A successful data reception is acknowledged with an Ack packet. The receiver notifies the host of a failed transmission by sending a Nak (No Acknowledge) packet.

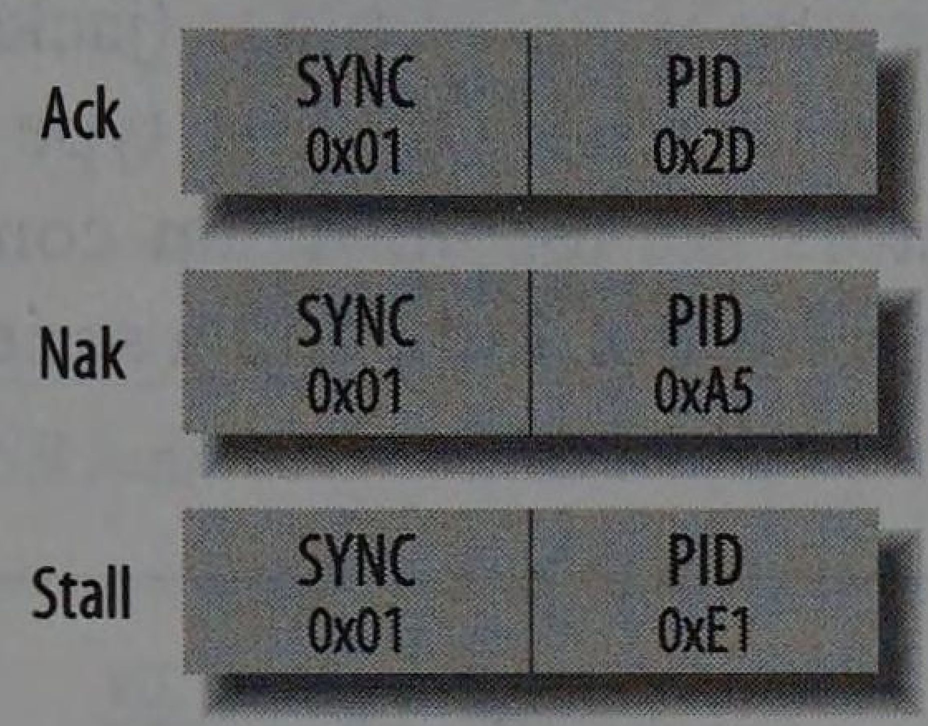


Figure 10-24. USB handshaking packets

A *descriptor* is a data packet used to inform the host of the capabilities of the device. It contains an identifier for the device's manufacturer, a product identifier, a class type, and the device's internal configuration, such as its power needs and end points. Each manufacturer has a unique ID, and each product in turn will also have a unique ID. Software on the host computer uses information obtained from a descriptor to determine what services a device can perform and how the host can interact with that device.

Full details of the USB protocols may be found in the USB technical documentation available from the USB web site (*http://www.usb.org*).

Physical Interface

USB uses a shielded, four-wire cable to interconnect devices on the network (Table 10-3). Data transmission is over a differential twisted pair (much like RS-422/485), labeled **D+** and **D-**. The other two wires are **VBUS**, which carries power to USB devices, and **GND**. Devices that use USB power are known as *bus-powered devices*, while those with their own external power supply are known as *self-powered devices*. To avoid confusion, the wires within a USB cable are color-coded.

Table 10-3. USB wires

Connector pin	Signal	Purpose	Wire color
1	VBUS	USB device power (+5V)	Red
3	D+	Differential data line	Green
2	D-	Differential data line	White
4	GND	Power and signal ground	Black

Some USB chips refer to **D+** and **D-** as **DP** and **DM**, respectively.

The connection from a device back to a host is known as an *upstream connection*. Similarly, connections from the host out to devices are known as *downstream connections*. Different connectors are used for upstream and downstream ports, with the specific intention of preventing loopback. The only way to connect a USB network is a tiered star. USB uses two types of plugs (jacks) and two types of receptacles (sockets) for cables and equipment. The first type is series A and is shown in Figure 10-25. Series A connectors are for upstream connections. In other words, a series A receptacle is found on a host or hub, and a series A plug is at the end of the cable that attaches to the host or hub.

Figure 10-25. Series A plug and receptacle

Series B connectors are shown in Figure 10-26. A series B receptacle is found on a USB device, and a series B plug is at the end of the cable coming downstream from a host or hub.

Figure 10-26. Series B plug and receptacle

Figure 10-27 shows how this works in practice. This ensures that USB devices, hosts/hubs, and USB cables are always connected in the right way. It is not possible to have a cable plugged in the wrong way or to directly connect two USB devices together.

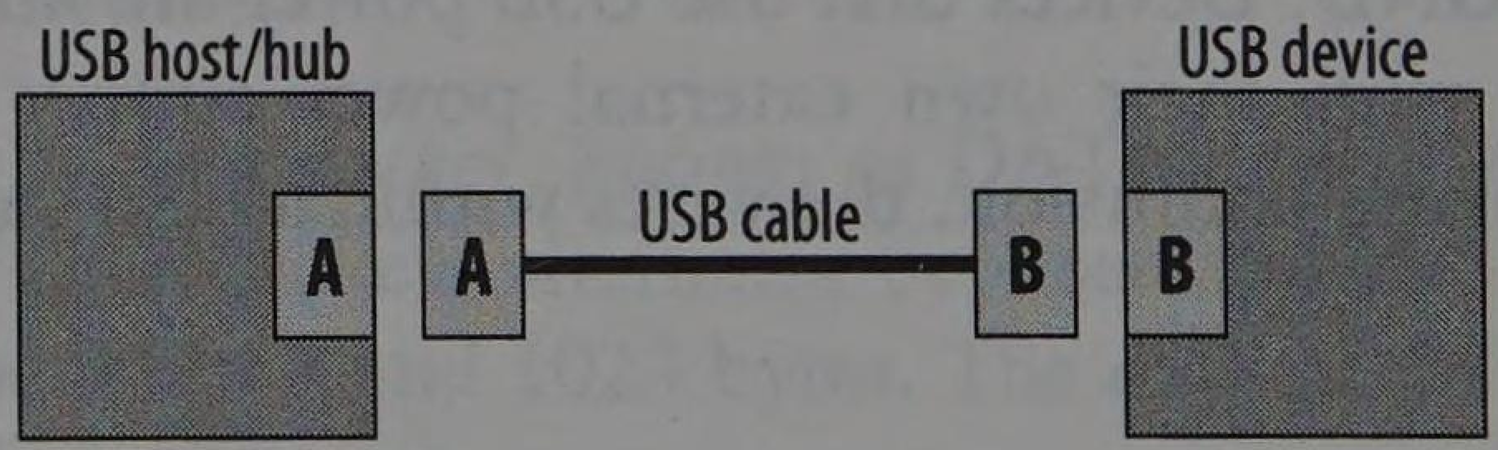


Figure 10-27. USB connectors and cable

Since a hub will be connected to USB devices downstream and a USB host or hub upstream, it will have both types of receptacle (Figure 10-28).

Chips that implement a USB interface require very few external components for the USB port. The schematic for an upstream port is shown in Figure 10-29.

In this example, the embedded system is powered from the USB port. If the embedded computer has its own power source, then no connection is made between VCC

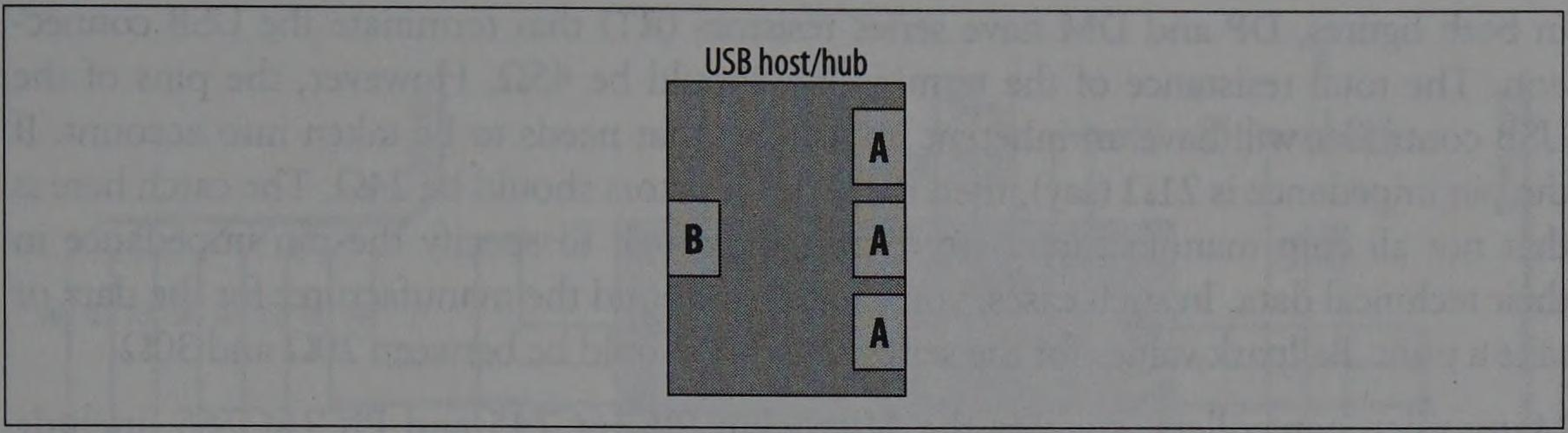


Figure 10-28. Receptacles on a USB hub

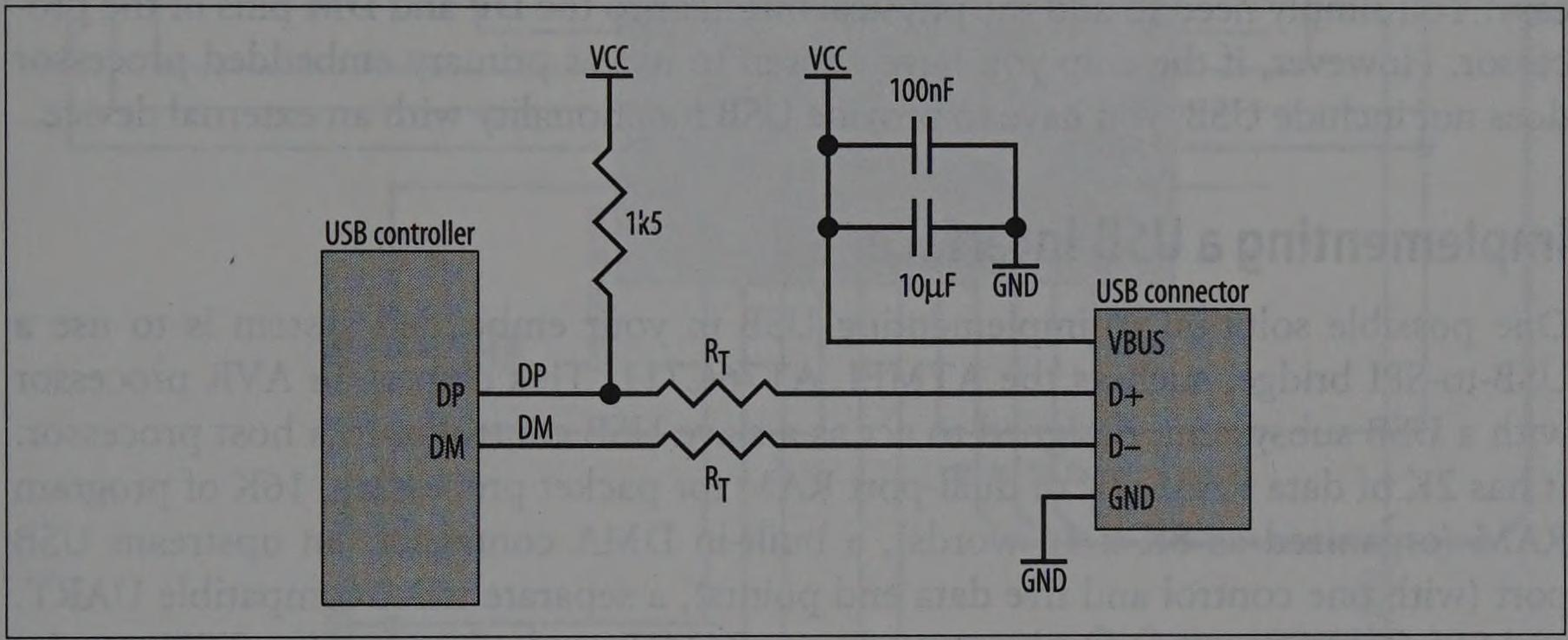


Figure 10-29. Upstream USB port

and pin 1 (**VBUS**) of the USB connector. The pull-up resistor connected to **DP** is required only on upstream ports. If you are implementing downstream ports on a hub, the pull-up is not required. However, downstream ports require pull-down resistors on both **DP** and **DM** (Figure 10-30).

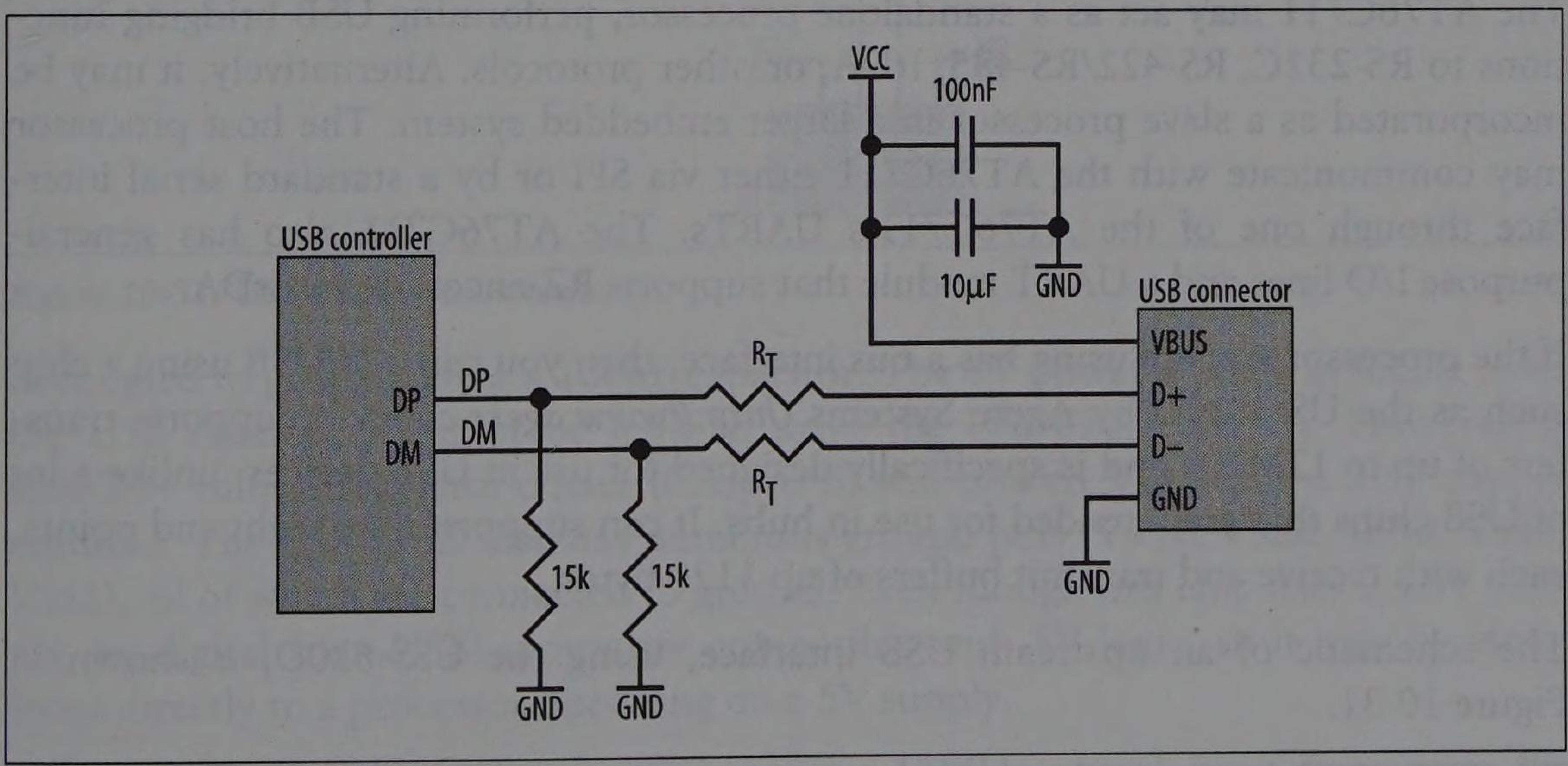


Figure 10-30. Downstream USB port

In both figures, **DP** and **DM** have series resistors (RT) that terminate the USB connection. The total resistance of the termination should be 45Ω. However, the pins of the USB controller will have an inherent impedance that needs to be taken into account. If the pin impedance is 21Ω (say), then the series resistors should be 24Ω. The catch here is that not all chip manufacturers are thorough enough to specify the pin impedance in their technical data. In such cases, you can either hound the manufacturer for the data or take a punt. Ballpark values for the series resistors should be between 20Ω and 30Ω.

Many microcontrollers, such as the Microchip PIC16C745 and PIC16C765, include USB modules as part of their suite of I/O. Implementing USB with such processors is easy. You simply need to add the physical interface to the **DP** and **DM** pins of the processor. However, if the chip you have chosen to use as primary embedded processor does not include USB, you have to provide USB functionality with an external device.

Implementing a USB Interface

One possible solution to implementing USB in your embedded system is to use a USB-to-SPI bridge, such as the ATMEL AT76C711. This chip is an AVR processor with a USB subsystem, designed to act as a slave USB controller to a host processor. It has 2K of data RAM, 2K of dual-port RAM for packet processing, 16K of program RAM (organized as 8K x 16 words), a built-in DMA controller, an upstream USB port (with one control and five data end points), a separate IrDA-compatible UART, and SPI. The processor may be run at up to 24MHz and operates off a 3.3V supply. At reset, the AT76C711 automatically loads its software from an external AT45DBxxx DataFlash (Chapter 9) to the program RAM. Since the AT76C711's program space is quite small, one of the smaller AT45DBxxx DataFlashes will be sufficient. Alternatively, a host processor may load the AT76C711's code directly into its program RAM while it is held in reset.

The AT76C711 may act as a standalone processor, performing USB bridging functions to RS-232C, RS-422/RS-485, IrDA, or other protocols. Alternatively, it may be incorporated as a slave processor in a larger embedded system. The host processor may communicate with the AT76C711 either via SPI or by a standard serial interface through one of the AT76C711's UARTs. The AT76C711 also has general-purpose I/O lines and a UART module that supports RZ encoding for IrDA.

If the processor you are using has a bus interface, then you can add USB using a chip such as the USS-820D by Agere Systems (*http://www.agere.com*). It supports transfers of up to 12Mbps and is specifically designed for use in USB devices, unlike a lot of USB chips that are intended for use in hubs. It can support up to eight end points, each with receive and transmit buffers of up 1120 bytes.

The schematic of an upstream USB interface, using the USS-820D, is shown in Figure 10-31.

The USS-820D has several power-supply inputs (**VDDA**, **VDDT**, **VDD0**, **VDD1**), all of which operate from a 3.3V supply (**VDD** in the schematic). Each power pin is

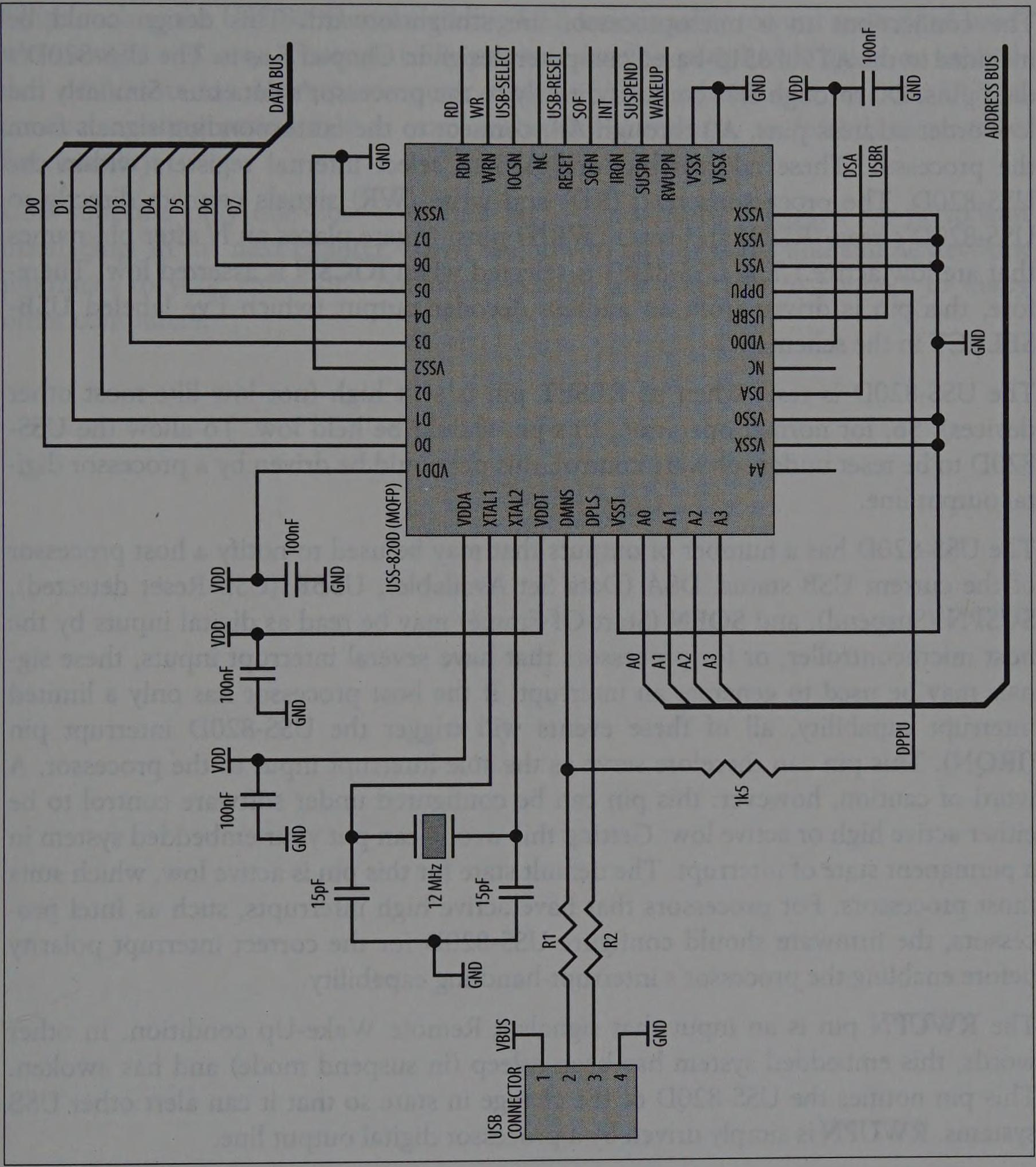


Figure 10-31. USS-820D USB interface

decoupled to ground using a 100nF capacitor. The 5V power (**VBUS**) available from the USB connector cannot be used to drive the USS-820D directly. However, a MAX604 voltage regulator circuit (Chapter 3) will convert **VBUS** to the 3.3V supply required. The USS-820D also has numerous ground pins (**VSST**, **VSSX**, **VSS0**, **VSS1**, **VSS2**), all of which are connected to ground. Even though this chip uses a 3.3V supply, its digital (non-USB) inputs are compatible with 5V logic, so it may be interfaced directly to a processor operating on a 5V supply.

XTAL1 and **XTAL2** are the connections for a 12MHz crystal, providing timing for the USB controller.

The connections to a microprocessor are straightforward. This design could be included in the AT90S8515-based computer design in Chapter 6 as is. The USS-820D's data pins, **D0** through **D7**, connect directly to the processor's data bus. Similarly the low-order address pins, **A0** through **A4**, connect to the corresponding signals from the processor. These address bits are used to select internal registers within the USS-820D. The processor's read ($\overline{\mathbf{RD}}$) and write ($\overline{\mathbf{WR}}$) signals connect directly to USS-820D's read ($\overline{\mathbf{RDN}}$) and write ($\overline{\mathbf{WRN}}$) pins. (Agere places an *N* after pin names that are low active.) The USS-820D is selected when $\overline{\mathbf{IOCSN}}$ is asserted low. Therefore, this pin is driven from an address decoder output (which I've labeled **USB-SELECT** in the schematic).

The USS-820D is reset when its **RESET** pin is sent high (not low like most other devices). So, for normal operation, this pin should be held low. To allow the USS-820D to be reset under software control, this pin could be driven by a processor digital output line.

The USS-820D has a number of outputs that may be used to notify a host processor of the current USB status. **DSA** (Data Set Available), **USBR** (USB Reset detected), $\overline{\mathbf{SUSPN}}$ (Suspend), and **SOFN** (Start Of Frame) may be read as digital inputs by the host microcontroller, or for processors that have several interrupt inputs, these signals may be used to generate an interrupt. If the host processor has only a limited interrupt capability, all of these events will trigger the USS-820D interrupt pin (**IRQN**). This pin can therefore serve as the sole interrupt input to the processor. A word of caution, however: this pin can be configured under software control to be either active high or active low. Getting this wrong can put your embedded system in a permanent state of interrupt. The default state for this pin is active low, which suits most processors. For processors that have active high interrupts, such as Intel processors, the firmware should configure USS-820D for the correct interrupt polarity before enabling the processor's interrupt-handling capability.

The $\overline{\mathbf{RWUPN}}$ pin is an input that signals a Remote Wake-Up condition. In other words, this embedded system has been asleep (in suspend mode) and has awoken. This pin notifies the USS-820D of the change in state so that it can alert other USB systems. $\overline{\mathbf{RWUPN}}$ is simply driven by a processor digital output line.

The USB differential data signals are pins **DPLS** (Data Plus, **D+**) and **DMNS** (Data Minus, **D-**). These are connected to the USB connector through series-termination resistors. Agere Systems suggests a nominal value of 24Ω. For an upstream connection, **DPLS** (**D+**) requires a pull-up resistor of 1.5kΩ. Normally, this resistor is connected to +5V. However, the USS-820D provides a special pin (**DPPU**) specifically for this purpose. Thus, under software control, the USS-820D can simulate a USB device disconnect. It will appear to an upstream hub that the system containing the USS-820D has been physically disconnected, even though it is still attached. This can be useful during development and testing. It also allows the USB device to decide whether a host knows it is connected. **DPPU** may be decoupled to ground using a 10nF capacitor.

Chips such as the USS-820D make adding USB functionality to your embedded hardware simple and easy. Through USB, you can develop peripherals based on embedded processors for desktop computer systems. Alternatively, you can use USB to connect existing peripherals to your embedded computer, to further increase its functionality.

USB supports only one host computer and is specifically intended for peripheral interfacing. In the next chapter, you'll see how to add low-cost and simple network interfaces to your embedded computer system, allowing you to connect to many other computers.

CHAPTER 11

Networks

Never let the future disturb you. You will meet it, if you have to, with the same weapons of reason that today arm you against the present.
—Marcus Aurelius Antoninus
Meditations

No town or freeman shall be compelled to build bridges . . . except those with an ancient obligation to do so.
—The Magna Carta

In this chapter, we'll look at connecting your embedded computer to the real world by adding a *Local Area Network* (*LAN*) interface. Of the wide variety of networks employed, some are very common, some not so common. We'll take a look at RS-485, CAN, and Ethernet.

RS-485, a simple network used for connecting small controllers, is very low in cost and simple to implement. CAN is a network for industrial applications in which a conventional network just won't do. CAN is suited to electrically noisy and harsh conditions and is the network of choice in electrically severe environments. Ethernet is the intranet network that connects the world's desktop computers, as well as a host of other devices such as routers, gateways, printers, and other peripherals.

RS-485

RS-485 is a variation on RS-422 (Chapter 10) used for low-cost networking and is commonly used in many industrial applications. It is one of the simplest and easiest networks to implement. It allows multiple systems (*nodes*) to exchange data over a single twisted pair (Figure 11-1).

RS-485 is based on a master/slave architecture. All transactions are initiated by the master, and a slave will transmit only when specifically instructed to do so. Many

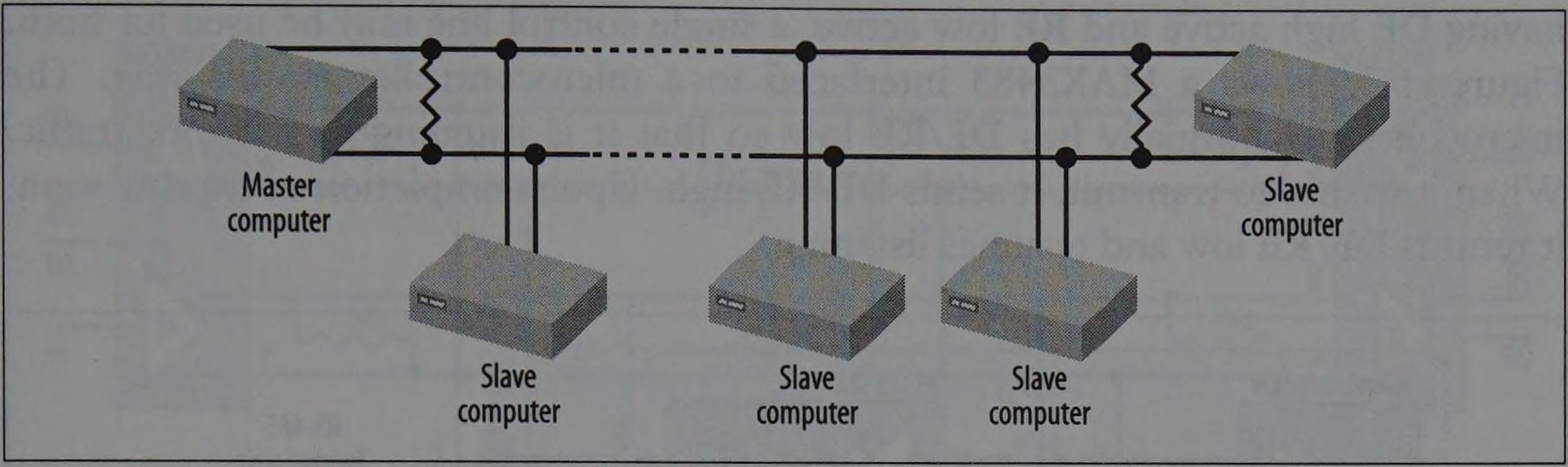


Figure 11-1. RS-485 network

different protocols run over RS-485, and often people will do their own thing and create their protocol specific to the application at hand.

The interface to the RS-485 network is provided by a transceiver, such as a Maxim MAX3483 (Figure 11-2).

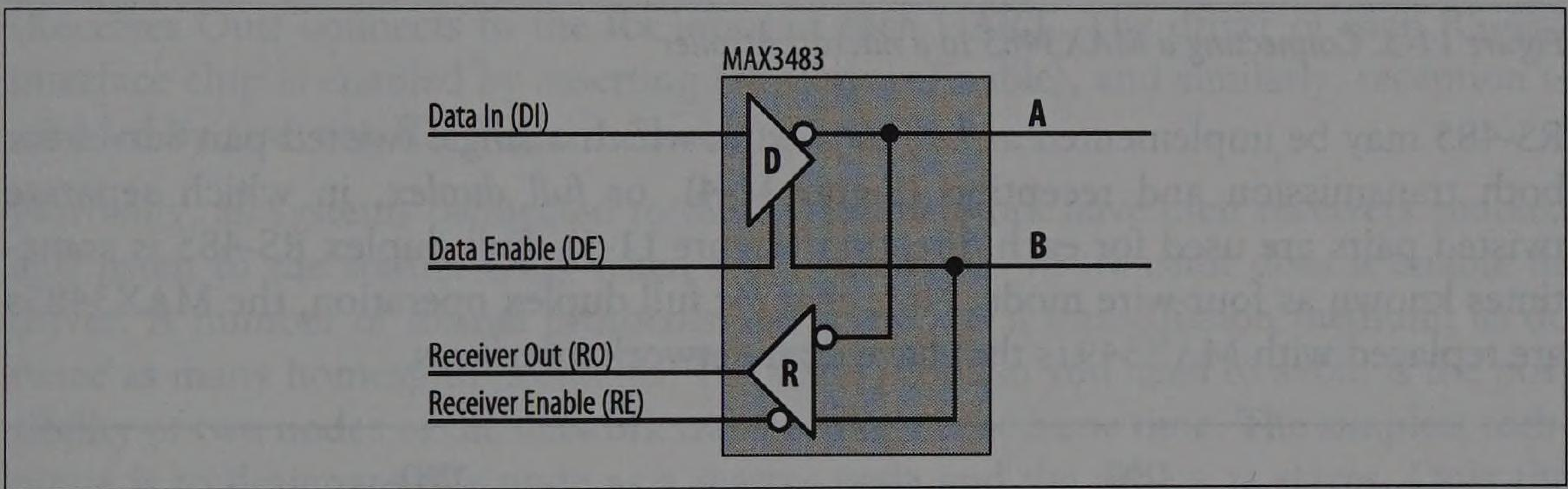


Figure 11-2. RS-485 transceiver

It is simply an RS-422 transceiver with enable inputs; using it in a design is straightforward. On the network side, the MAX3483 has two signal lines, **A** and **B**. This is the twisted-pair (network cable) attachment point. The MAX3483 also has Data In (**DI**) and Receiver Out (**RO**). These are connected to the **Tx** and **Rx** signals of the UART (or microcontroller), respectively.

Since it is connected to a common network upon which it must both listen and transmit, it has two control inputs, Data Enable (**DE**) and Receiver Enable (**$\overline{\text{RE}}$**). A high input to **DE** allows the **DI** input to be transmitted on the network. A low input to **DE** disables the output of the transmitter. Similarly, a low input to **$\overline{\text{RE}}$** enables the receiver, and network traffic is passed through to **RO**. **DE** and **$\overline{\text{RE}}$** are normally controlled by an I/O line of the processor. Now, you'll notice that **DE** is high active and **$\overline{\text{RE}}$** is low active. This is not by chance. A node on the network won't be receiving traffic if it's transmitting and, conversely, won't be transmitting if it is receiving. Therefore, only either the transmitter or the receiver should be active at any one time. If the transmitter is on, the receiver should be off, and vice versa. The control for the transmitter is therefore the logical opposite of the control for the receiver. By

having **DE** high active and **RE** low active, a single control line may be used for both. Figure 11-3 shows a MAX3483 interfaced to a microcontroller in this way. The microcontroller normally has **DE/RE** low so that it is listening to network traffic. When it wishes to transmit, it sends **DE/RE** high. Upon completion of transmission, it returns **DE/RE** low and resumes listening.

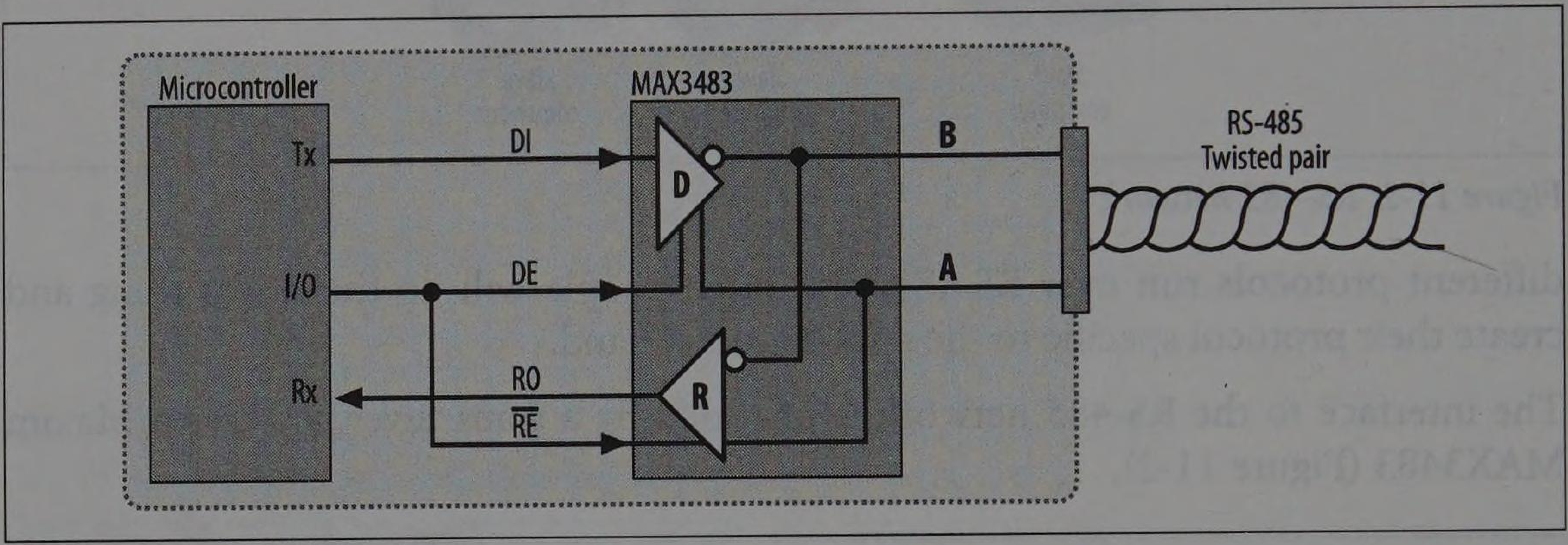


Figure 11-3. Connecting a MAX3483 to a microcontroller

RS-485 may be implemented as *half duplex*, in which a single twisted pair serves for both transmission and reception (Figure 11-4), or *full duplex*, in which separate twisted pairs are used for each direction (Figure 11-5). Full duplex RS-485 is sometimes known as four-wire mode. Note that for full duplex operation, the MAX3483s are replaced with MAX3491s that have dual network interfaces.

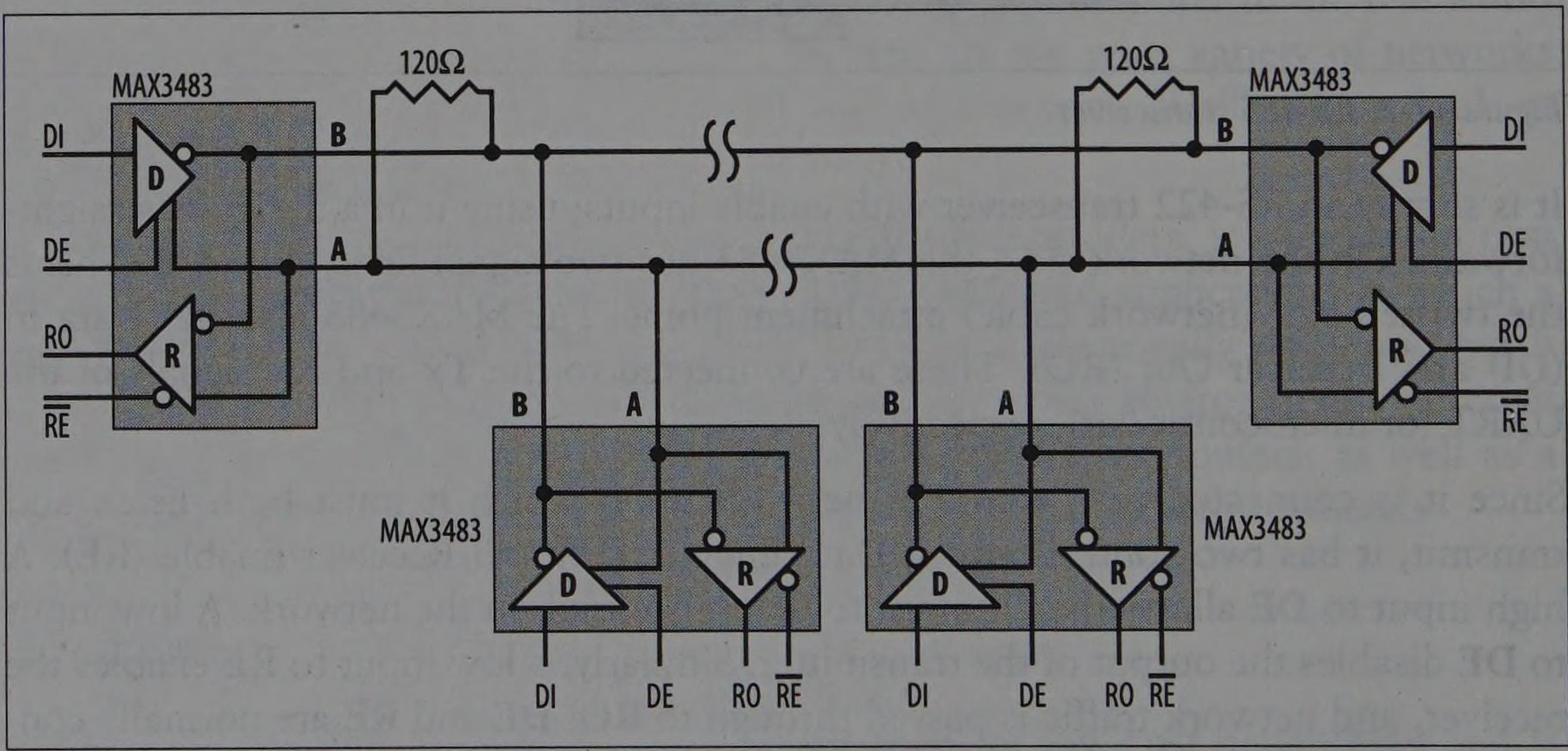


Figure 11-4. Half duplex RS-485

The two figures show four computers (nodes) connected to an RS-485 network. Each RS-485 interface chip (MAX3483 or MAX3491) exists in a separate embedded computer. The UART transmitter output, **Tx**, in each embedded system is connected to the respective **DI** (Driver In) of each of the RS-485 interface chips. Similarly, **RO**

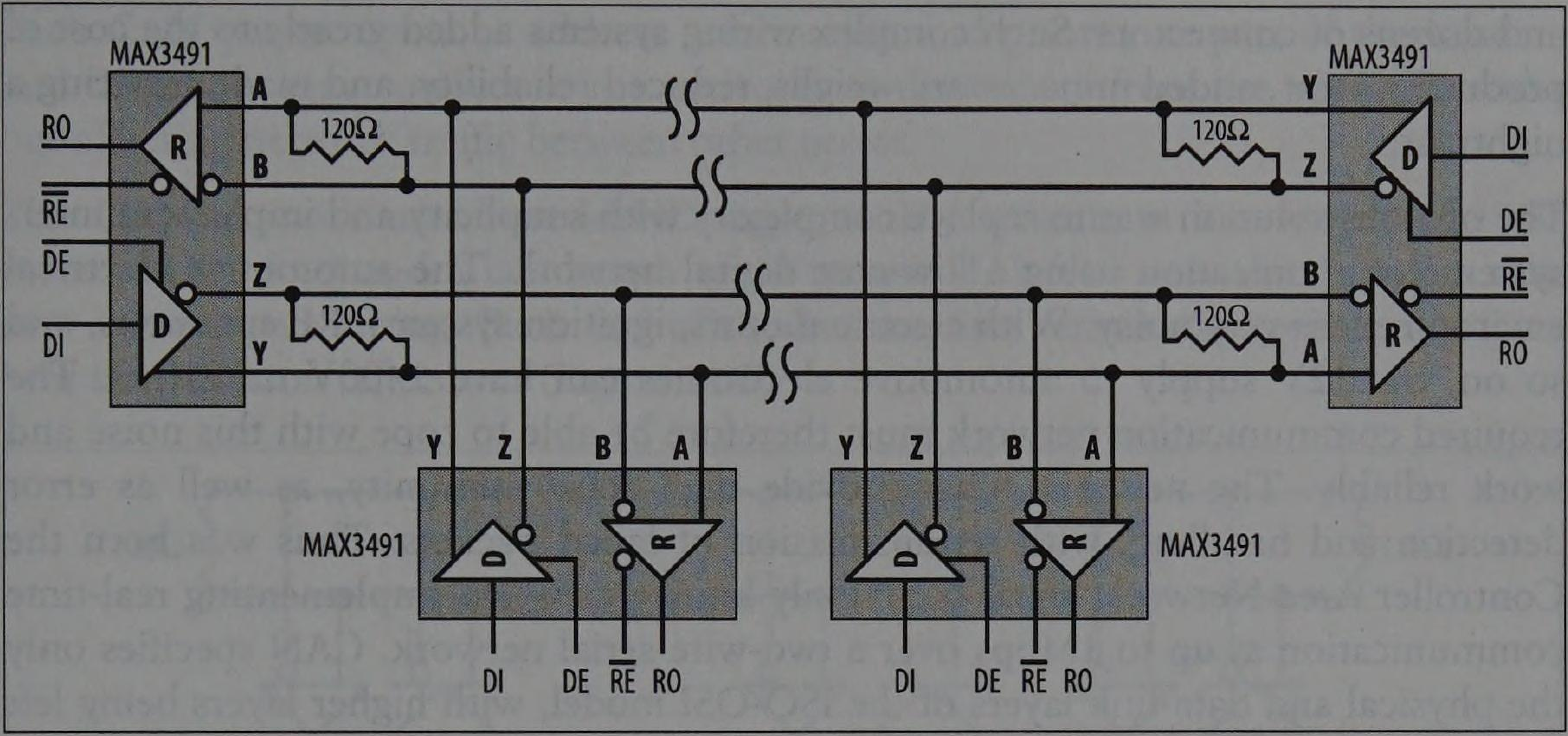


Figure 11-5. Full duplex RS-485

(Receiver Out) connects to the **Rx** input of each UART. The driver of each RS-485 interface chip is enabled by asserting **DE** (Driver Enable), and similarly, reception is enabled by asserting $\overline{\textbf{RE}}$ (Receive Enable).

Normally, all systems connected to the RS-485 network have their receivers enabled and listen to the traffic. Only when a system wishes to transmit does it enable its driver. A number of formal protocols use RS-485 as a transmission medium, as do twice as many homespun protocols. The main problem you need to avoid is the possibility of two nodes of the network transmitting at the same time. The simplest technique is to designate one node as a master node and the others as slaves. Only the master may initiate a transmission on the network, and a slave may respond directly only to the master, once that master has finished.

The number of nodes possible on the network is limited by the driving capability of the interface chips. Normally, this limit is 32 nodes per network, but some chips can support up to 512 nodes.

Controller Area Network (CAN)

Through the late '70s and '80s, the complexity of automotive electronics had grown considerably, with engine management systems, ABS braking, active suspension, electronic transmissions, and automated lighting, air-conditioning, security, and central locking. These individual systems do not exist in isolation; each is part of an integrated whole. A considerable amount of information exchange is required, and therefore some means of system interconnection must be provided. The conventional method was for point-to-point wiring, providing discrete interconnection between each subsystem. This methodology was a natural evolution from the simple electrical systems of earlier cars, but as automotive complexity grew, such a scheme proved vastly inadequate. Each car could have several kilometers worth of wiring

and dozens of connectors. Such complex wiring systems added greatly to the cost of producing a car, added unnecessary weight, reduced reliability, and made servicing a nightmare.

The obvious solution was to replace complexity with simplicity and implement intersystem communication using a low-cost digital network. The automotive electrical environment is very noisy. With electric motors, ignition systems, RF emissions, and so on, the 12V supply to automotive electronics can have ±400V transients. The required communication network must therefore be able to cope with this noise and work reliably. The network must provide high-noise immunity, as well as error detection and handling, with retransmission of failed packets. Thus was born the Controller Area Network, more commonly known as CAN, implementing real-time communication at up to 1Mbps over a two-wire serial network. CAN specifies only the physical and data-link layers of the ISO-OSI model, with higher layers being left to the specific implementation.

Bosch developed CAN in Europe in the late 1980s, originally for use in cars. Because of its robustness, CAN has expanded beyond its automotive origins and can now be found in industrial automation, trains, ship navigation and control systems, medical systems, photocopiers, agricultural machinery, household appliances, office automation, and elevators. CAN is now an international standard under ISO11898 and ISO11519-2.

CAN supports multiple masters on the network, with each master responsible for local sensing and control within the distributed system (Figure 11-6).

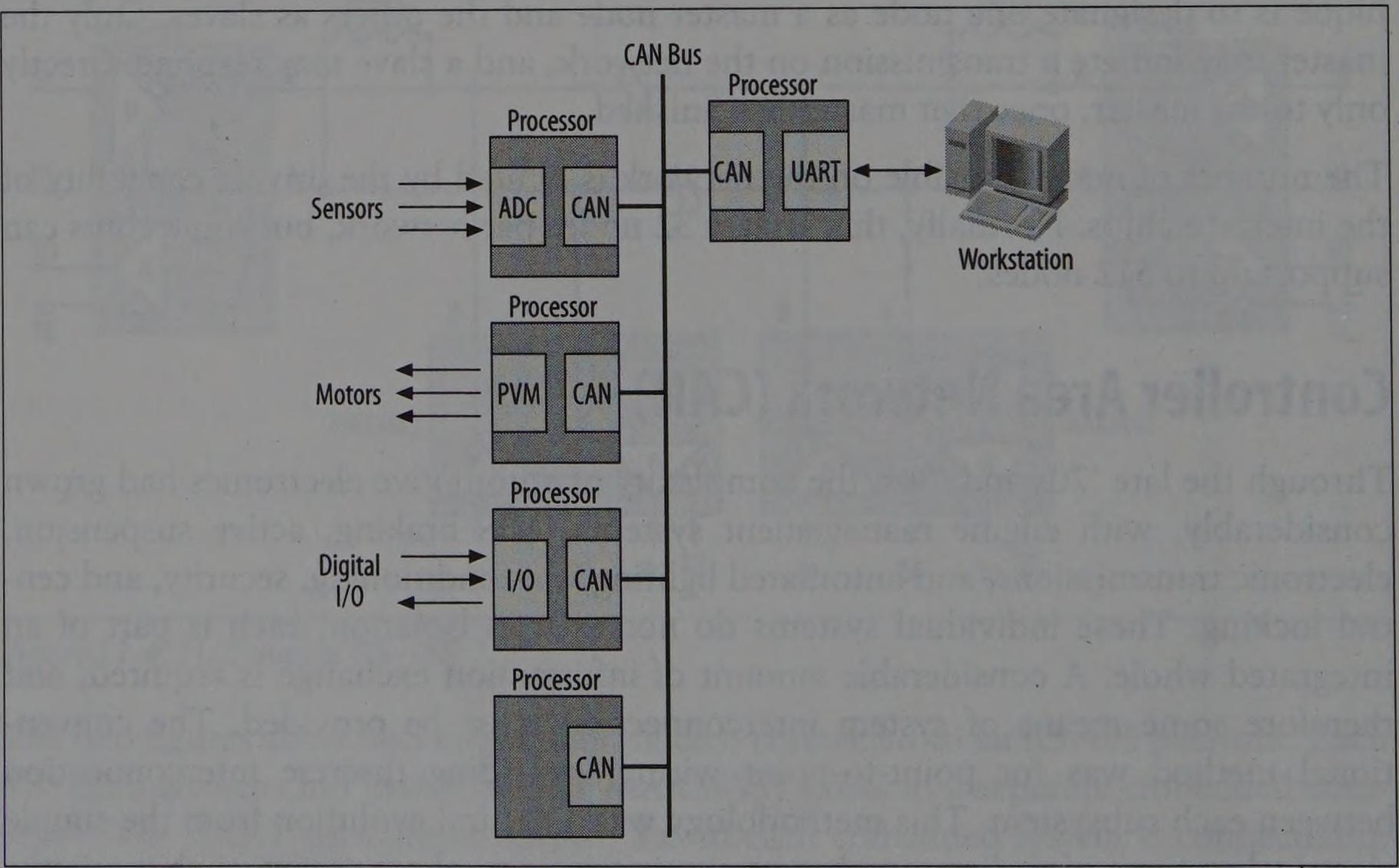


Figure 11-6. CAN distributed system

Each CAN packet contains address information and priority as part of the header, and the nodes may connect to the network, or disconnect from the network, without affecting network traffic between other nodes.

The CAN network uses wired-AND logic, with a maximum bus length of 1000 meters (3300 feet) and a bus length of 40 meters (133 feet) at maximum data rate over twisted-pair wiring. Each end of the bus requires termination resistors to prevent transmission reflections (Figure 11-7).

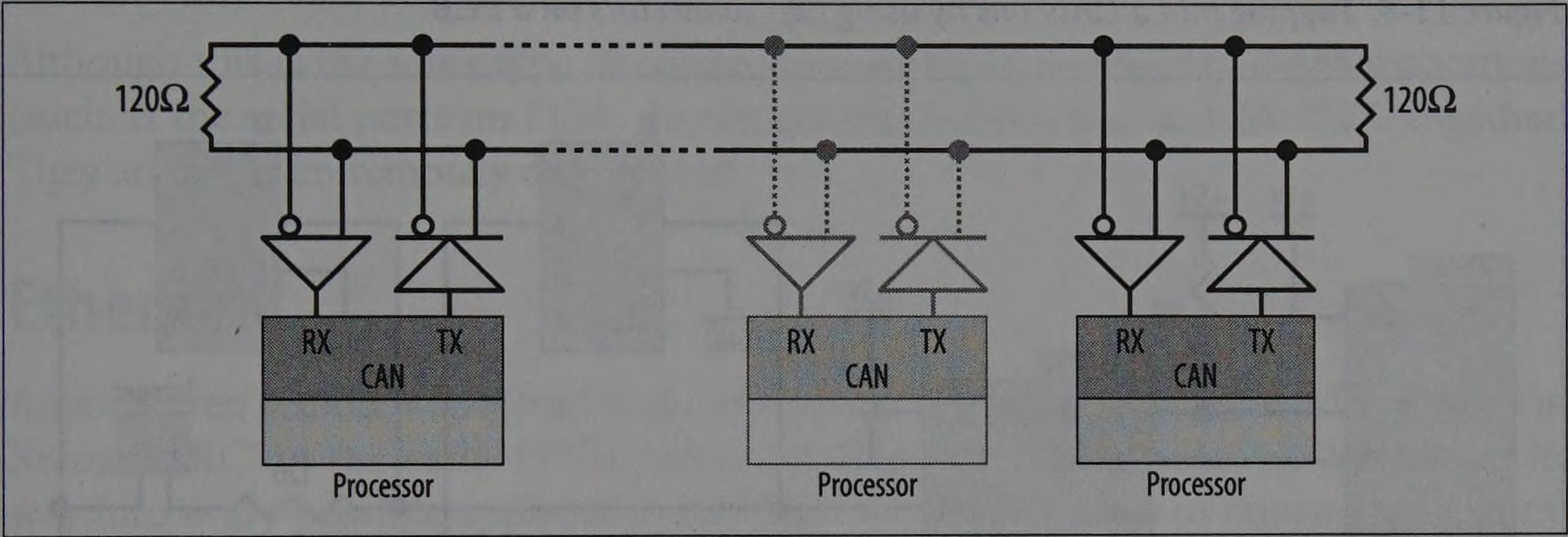


Figure 11-7. CAN bus

Many processors intended for use in harsh or electrically noisy industrial applications include a CAN module. A number of Philips microcontrollers include CAN, as do a few PICs. The DSP56805 processor we covered in Chapter 8 also has a CAN interface. For processors that do not include CAN, CAN interface modules are available. The Microchip MCP2510 provides a CAN module that interfaces to a host processor via SPI. Adding CAN to any embedded system is therefore a simple task.

Typically, a microprocessor that supports CAN will include a CAN interface module, which provides most of the functionality. The only additional support required is a CAN interface driver (just as in RS485 and RS232C). Philips Semiconductor produces a CAN driver, the PCA82C250T, which makes interfacing to the CAN bus very easy.

Your embedded computer must also have some way of physically attaching to the bus. The simplest method is simply to bring the bus into the computer system on one connector, tap off it, and then route it out through another connector (Figure 11-8).

To see how we can use CAN, let's look at the DSP56805 processor. This processor has a CAN network module as part of its suite of onboard peripherals. The schematic for interfacing a processor's CAN module to a CAN bus is shown in Figure 11-9.

The DSP56805 has two CAN interface signals, **MSCAN-TX** and **MSCAN-RX**, the CAN transmitter and receiver, respectively. These are connected to the PCA82C250T, which provides the interface to the CAN bus. Note that the DSP56805 requires a 3.3V supply, while the PCA82C250T requires a 5V supply. A pull-up resistor brings the **MSCAN-TX** output of the processor to the required logic

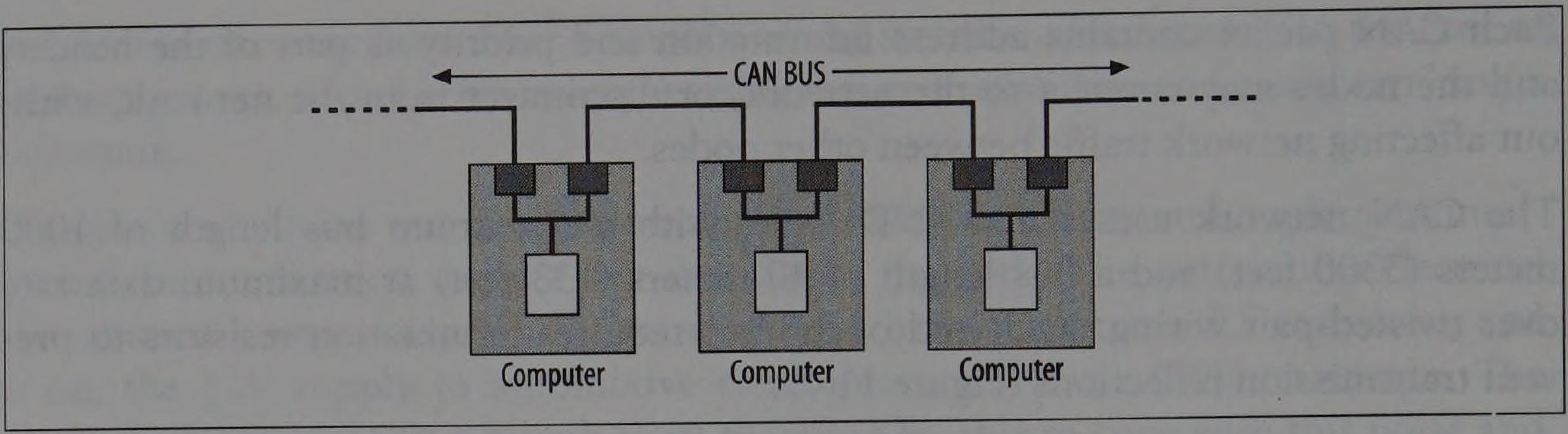


Figure 11-8. Tapping into a CAN bus by using two connectors on a PCB

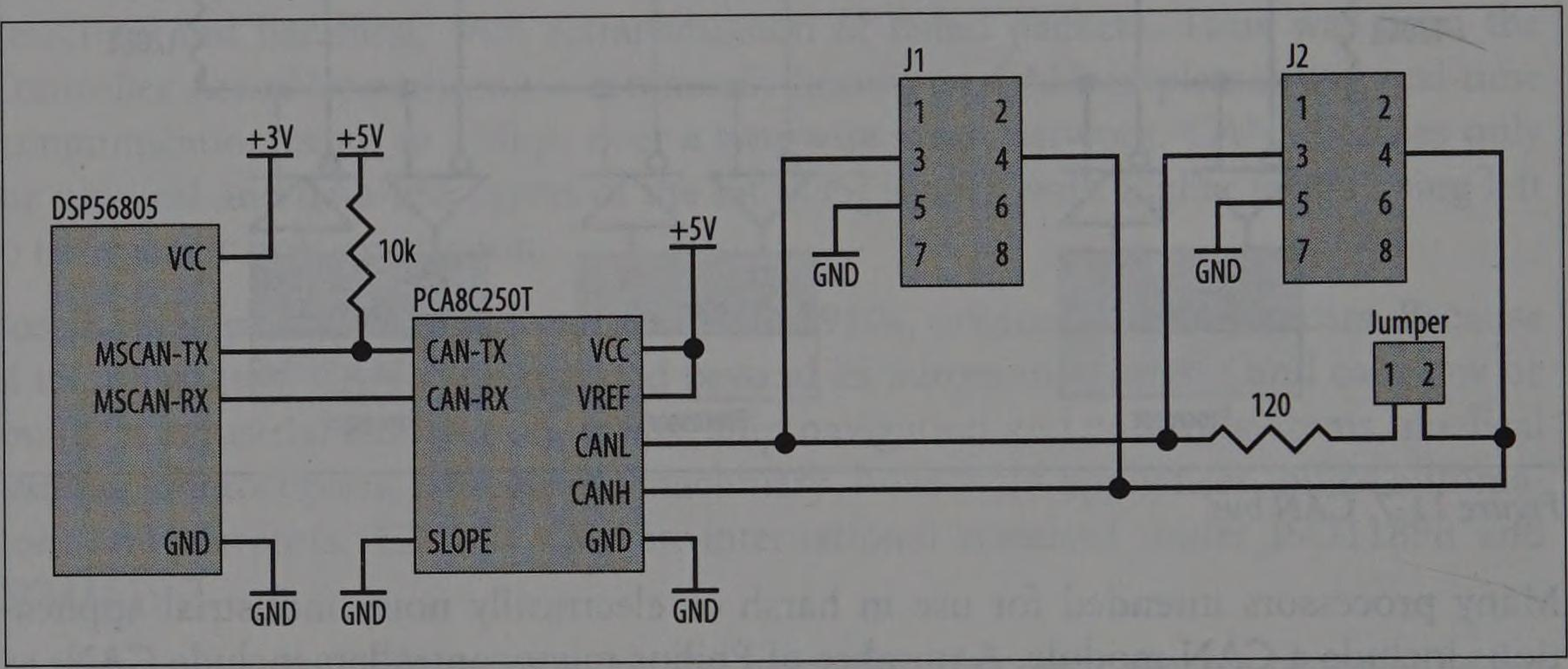


Figure 11-9. CAN interface for a DSP56805 processor

high level for the PCA82C250T. While CAN requires only two signal lines and ground, the actual connectors have eight pins. Since the CAN bus requires a termination resistor at each end, we provide a 120Ω resistor, should our computer be placed at the bus end. A jumper allows it to be brought in-circuit or disabled as needed. So, if our computer is at the end of the CAN bus, the jumper is closed and the bus is terminated. If our computer is not an end-point machine, the jumper is left open, and the resistor plays no part. Note that having a termination resistor active (jumper closed) when this computer is not at an end point is a good way to ensure an unreliable CAN bus! Resistors should be active at bus ends only.

Many implementations of CAN just use standard IDC-type headers for the connectors. However, the actual CAN standard specifies that the connector should be a nine-pin Sub-D connector. The pinouts for this connector are listed in Table 11-1.

Table 11-1. CAN pinouts

Pin	Signal/Use
1	Reserved
2	CAN_L
3	Ground
4	Reserved

Table 11-1. CAN pinouts (continued)

Pin	Signal/Use
5	Reserved
6	Ground
7	CAN_H
8	Reserved
9	V+ (optional power source)

Although this is the same type of connector used in some RS-232C implementations (such as the serial ports on PCs), do not connect a CAN bus and RS-232C together. They are not even remotely compatible!

Ethernet

Anyone even remotely involved with computers has heard of Ethernet. Developed at Xerox PARC* in the early 1970s, this local area networking standard has found its way into every possible application and has evolved over time to encompass a number of standards ranging from wireless networks (802.11) to gigabit Ethernet.

In this section, I'll look at how you add a simple Ethernet interface to your embedded computer. We will develop a 10Mbps interface only, as higher-speed interfaces require special attention to PCB design and EMC issues. So, for your sake of ease and reliability, we'll keep it simple and low speed.

The Ethernet standards and protocols are detailed in *Ethernet: The Definitive Guide* by Charles E. Spurgeon, available from O'Reilly & Associates. This excellent book gives definitive coverage of Ethernet and is a must for anyone developing Ethernet-based hardware. It is essential background reading.

By adding Ethernet to your embedded system, you gain access to a network and all the possibilities that brings. You can send data to a host computer at high speed and access printers, file servers, databases, and even the Internet. You can also monitor and control your embedded system from afar or even have it send you email when it needs attention. Take an AT90S8515 AVR and add an Ethernet interface and some high-capacity flash memory, and you have yourself a simple web server. Add an ADC and some sensors, and your web server becomes a weather station showing current or past conditions to anyone on the Internet. Use a higher-speed processor, several Ethernet ports, and the appropriate software, and you have yourself a simple gateway or firewall. You could even build an Ethernet-to-Ethernet (or serial, parallel port, or USB) bridge. The possibilities are limited only by your imagination.

* PARC is the Palo Alto Research Center (*http://www.parc.com*). For an interesting history of PARC (and the computer industry in general), read Robert X. Cringley's *Accidental Empires*.

There was a time when developing an Ethernet interface was a major exercise. These were complicated circuits, using lots of chips and hundreds of support components. An Ethernet interface could fill a moderate PCB all on its own. Not any more. In these days of large-scale integration, adding Ethernet to your design is easy, as we will see.

Adding an Ethernet Interface

Crystal Semiconductor, now part of Cirrus Logic (*http://www.cirrus.com*), produces a single-chip Ethernet controller, known as the CS8900A. This chip allows you to add a simple (and low-cost) 10Mbps Ethernet interface to your embedded system. Full documentation on this chip is available from the Cirrus Logic web site. As the CS8900A is a commonly used Ethernet controller, plenty of source code is available on the Internet. Just use your favorite search engine to hunt it down. When you design a system based on the CS8900A, you can actually email your design to the engineers at Cirrus Logic, and they will check it out for you, offering advice and pointing out mistakes. The email address for this service is *ethernet@crystal.cirrus.com*.

The CS8900A supports 10BASE-2, 10BASE-T, and AUI (Attachment Unit Interface) Ethernet ports. 10BASE-T and 100BASE-T are by far the most common types of Ethernet interface, supporting data rates of 10Mbps and 100Mbps, respectively. Your desktop computer's Ethernet interface is most likely a 10/100BASE-T port with an eight-pin RJ-45 connector. (RJ-45 connectors look like, but are not the same as, standard telephone jacks.) The cabling used is *UTP* (Unshielded Twisted Pair) Category 5 cable, more commonly known simply as *CAT5*. Just like RS-422, RS-485, USB, and CAN, 10/100BASE-T Ethernet transmits using balanced differential signals. Four wires are used: two for the transmitter pair and two for the receiver pair. One wire of the pair carries a signal voltage of 0 to +2.5V, while the other wire carries a voltage of 0 to -2.5V, giving a signal difference of 5Vpp.

Table 11-2 shows the pin connections for an RJ-45 connector. The wires within the CAT5 cable are color-coded for easy identification.

Table 11-2. RJ-45 connector signals

Pin	Signal name	Purpose	Wire color
1	TD+	Transmitted data	White/orange
2	TD-	Transmitted data	Orange
3	RD+	Received data	White/green
4	NC	No connection	Blue
5	NC	No connection	White/blue
6	RD-	Received data	Green
7	NC	No connection	White/brown
8	NC	No connection	Brown

A block diagram of a CS8900A implementation is shown in Figure 11-10.

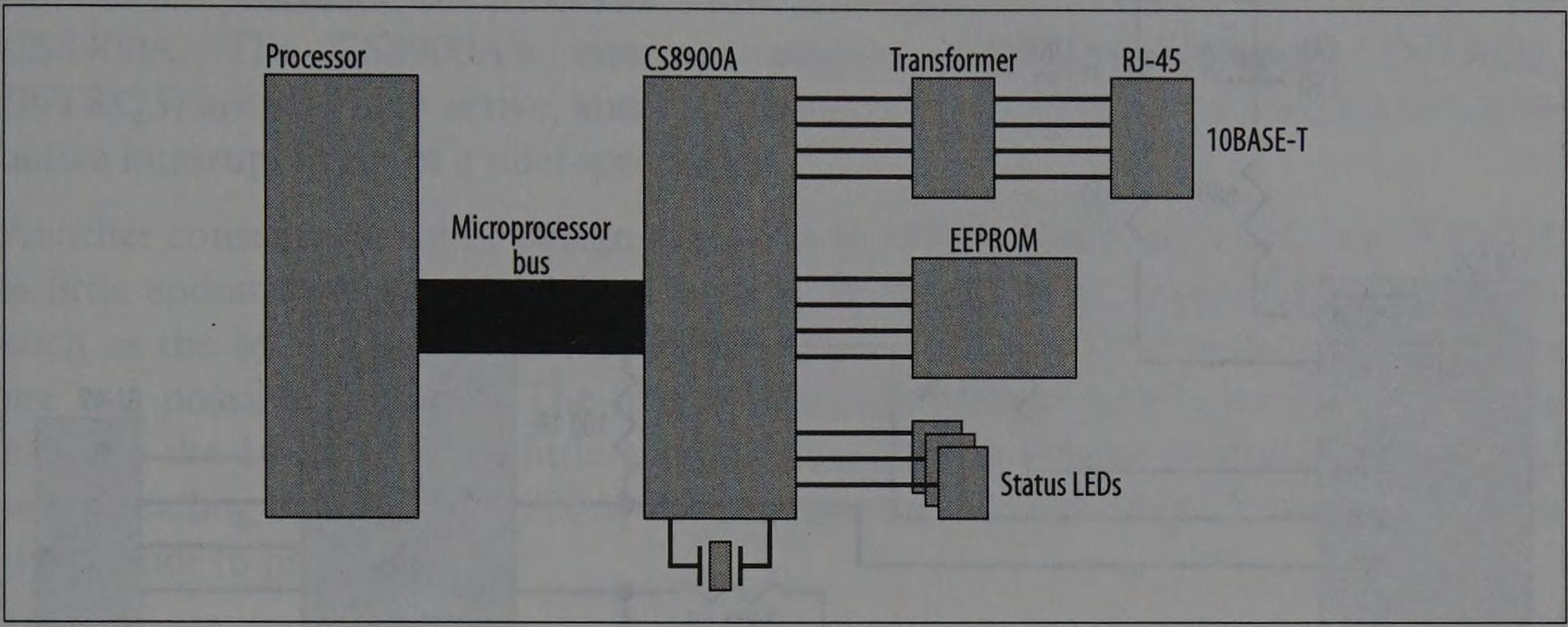


Figure 11-10. Block diagram of a CS8900A implementation

As the CS8900A has 100 pins and several different modes of operation, I won't show you an entire schematic in one hit. Instead, I'll work through each stage of a CS8900A's design and explain its functionality and use as I go. This discussion will be targeted at a small embedded application. Some of the more complicated aspects of the CS8900A, which are applicable to desktop PCs, I will leave alone.

The CS8900A is connected to its 10BASE-T port through an isolation transformer. This transformer must have a winding ratio of 1:1 for the receiver and a winding ratio of 1:1.41 for the transmitter, if the CS8900A is used with a 5V supply. If used with a 3.3V supply, then the transformer's winding ratio for the transmitter must be 1:2.5. A number of manufacturers—such as Valor, PCA, YCL, and Bel—make isolation transformers (packaged as chips) with these winding ratios. The transmitter requires series-termination resistors of 24.9Ω, ±1%. The transmitter differential pair must be decoupled with each other using a 68pF capacitor. A 100Ω resistor (±1%) is required in parallel between the receiver's differential pair. The CS8900A can also directly drive LEDs, indicating Ethernet link status, and bus and network activity. The CS8900A has an additional pin (**RES**) that requires a 4.99kΩ (±1%) pull-down resistor. Figure 11-11 shows the CS8900A connected to a 10BASE-T port.

An external 20MHz crystal provides timing for the CS8900A. The crystal is connected across the **XTAL1** and **XTAL2** pins, and each pin is bypassed to ground using 33pF capacitors (Figure 11-12).

This Ethernet chip supports the 16-bit ISA bus architecture, the expansion bus found in older model PCs. However, ISA can easily be adapted to work with a range of non-ISA processors. The CS8900A may therefore be implemented in a variety of computer systems without difficulty. The CS8900A also supports operation in 8-bit mode and so can also be interfaced to microcontrollers with an 8-bit data bus, such as the AT90S8515 AVR. The CS8900A's input $\overline{\textbf{SBHE}}$ is used to place the chip in 16-bit mode operation after reset. Any activity on $\overline{\textbf{SBHE}}$ will place the CS8900A in

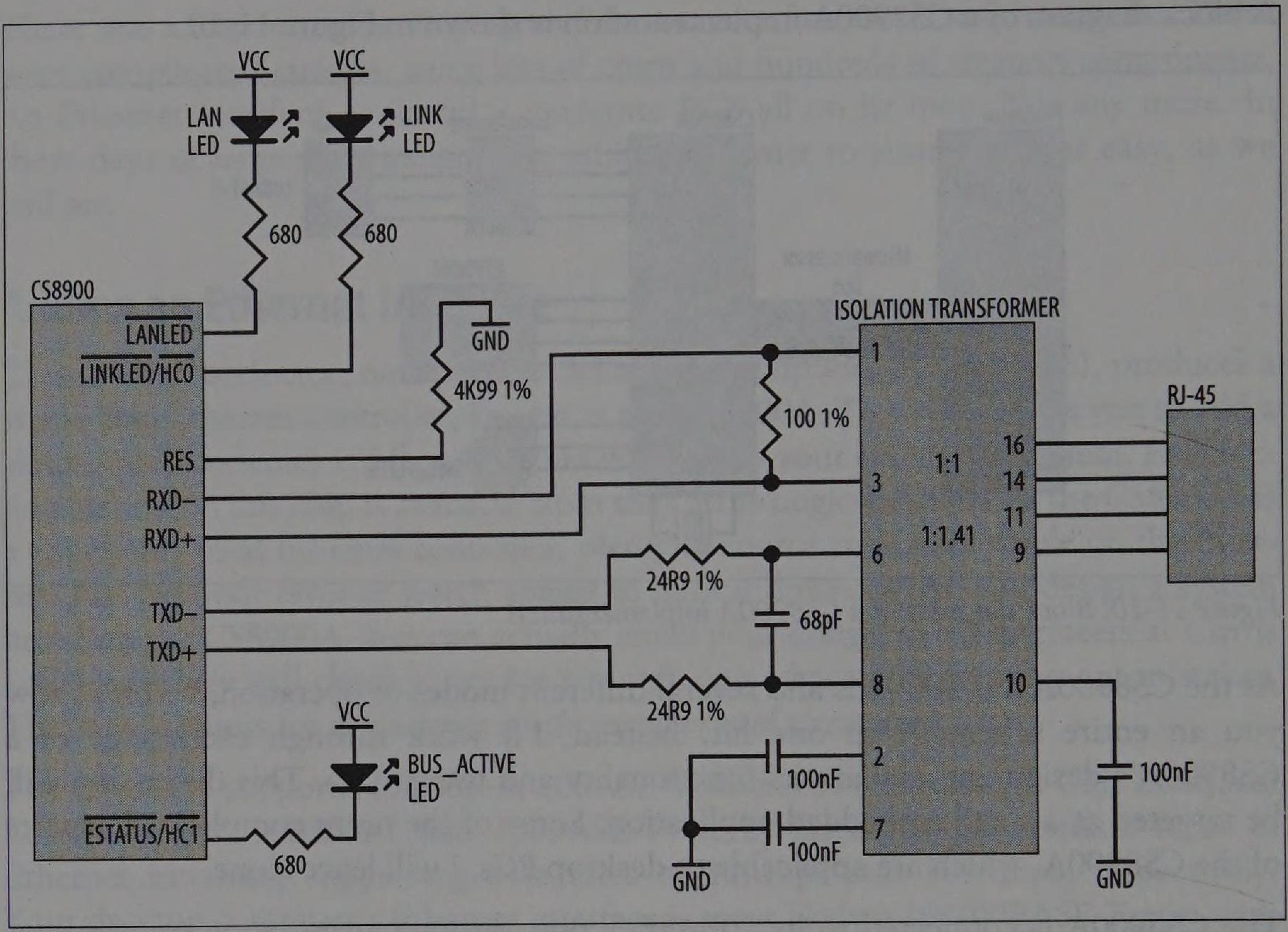


Figure 11-11. BASE-T interface

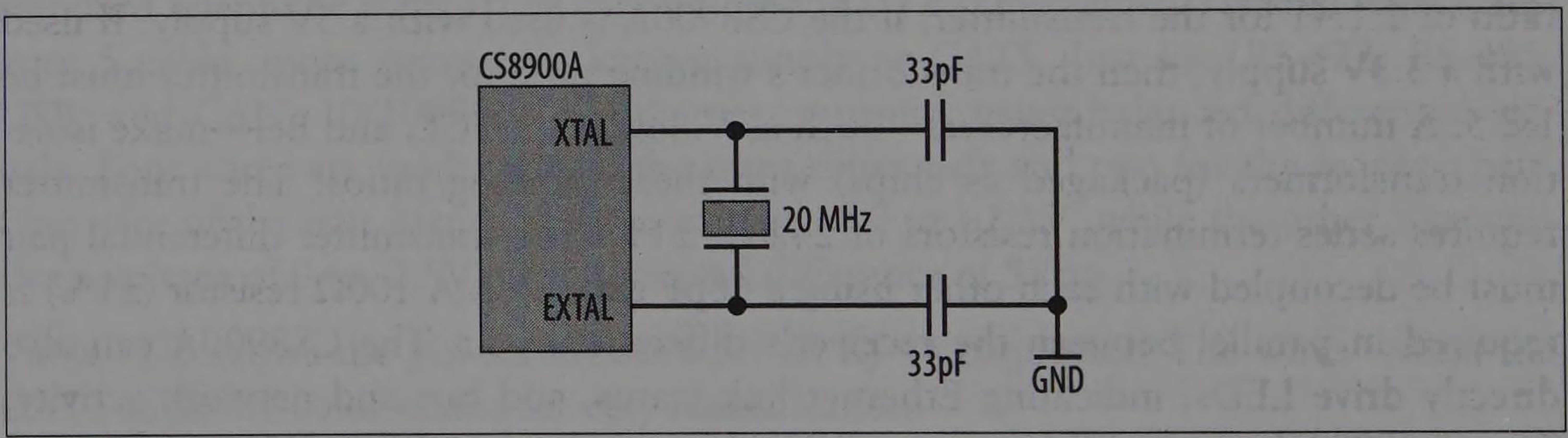


Figure 11-12. Crystal connections for the CS8900A

16-bit mode. The easiest way to ensure that there is activity of this input is simply to connect **$\overline{\text{SBHE}}$** to the processor's address line, **A0**. As soon as the processor begins to use its bus, the activity will place the CS8900A in 16-bit mode. For 8-bit operation, **$\overline{\text{SBHE}}$** is tied to ground. When used in 8-bit mode, interrupts are disabled and the CS8900A's status must be polled by software.

Before we look at the processor interface of the CS8900A, we need to note some important characteristics. On the CS8900A, **RESET** is active high. This can catch an unwary designer used to low-active resets. That **RESET** is high active derives from the fact that this chip was designed principally for use in PCs, as Intel processors also have a high-active reset. The CS8900A's reset may be driven by a digital output of a microcontroller so that it can be reset under software control. Alternatively, in

systems in which the CS8900A is to have a hardware-generated reset at the same time as the processor, the processor's low-active reset signal must be inverted for the CS8900A. The CS8900A's interrupt outputs (**INTRQ0**, **INTRQ1**, **INTRQ2**, **INTRQ3**) are also high active, and each must be inverted before connecting to a low-active interrupt input of a microprocessor.

Another consequence of its design for use in Intel-based systems is that the CS8900A is little endian in operation. When used in 16-bit mode with big-endian processors such as the MC68000 or the DSP56805, this endian difference is important. There are two possible solutions. The first is to simply byte-swap in software. Your code changes the 16-bit word to little-endian format before writing to the CS8900A. And when reading from the CS8900A, the processor must byte-swap the retrieved 16-bit word prior to processing.

However, there is an old saying that you should never fix in software what you can correct in hardware. The second solution is simply to byte-swap the data bus between the processor and the CS8900A. **D0**:**D7** of the processor are connected to **D8**:**D15** of the CS8900A, and **D8**:**D15** of the processor similarly go to **D0**:**D7** of the CS8900A. In this way, the endian-ness is reversed by the actual circuit board, and the software never needs to know the difference (Figure 11-13).

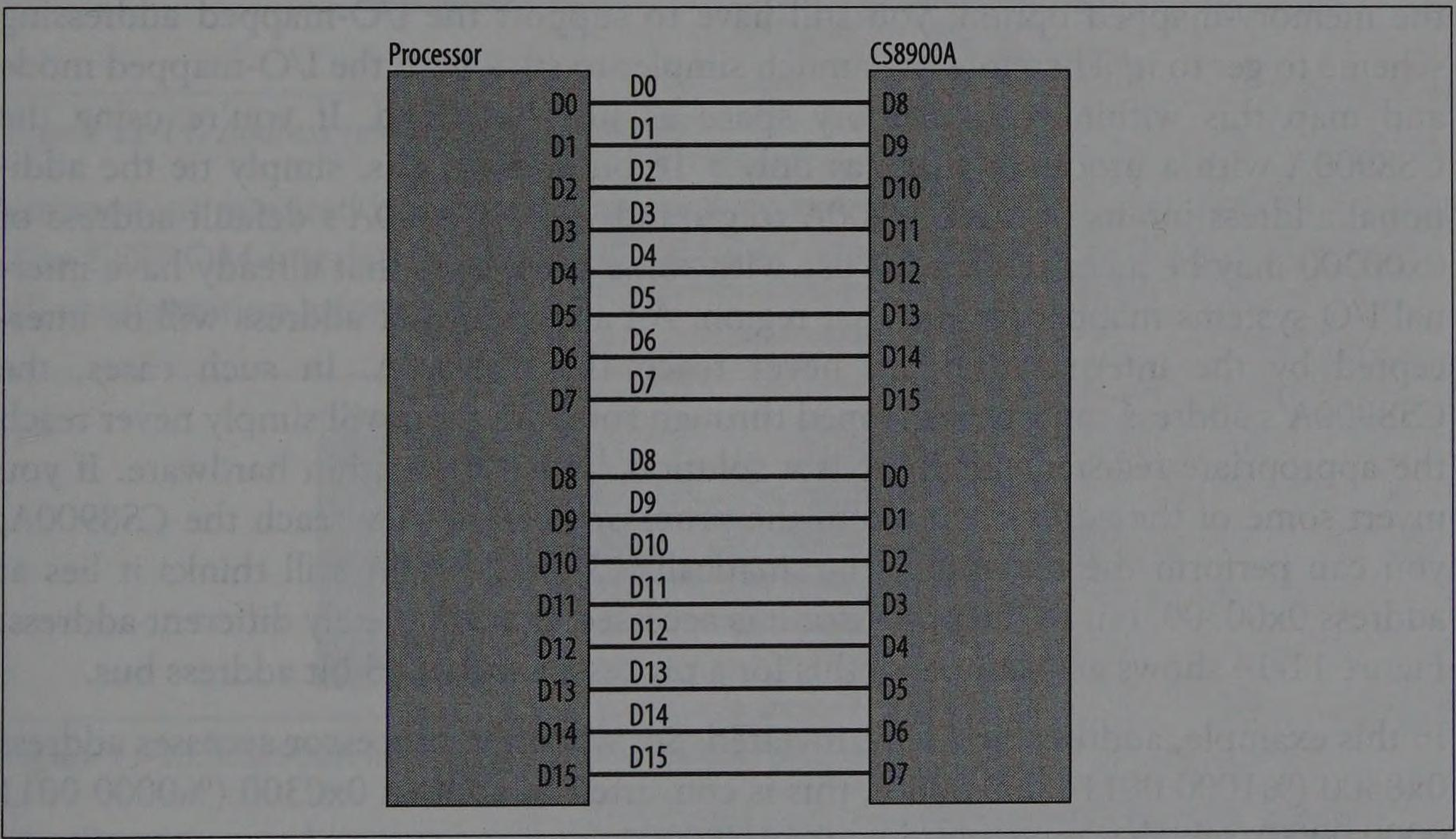


Figure 11-13. Endian swapping in hardware

The CS8900A has 20 address inputs. This may seem like a lot of address inputs for a peripheral, and it is. However, there is a reason. The CS8900A is principally an ISA-bus device, and the ISA bus supports separate memory and I/O memory spaces. Hence, the CS8900A has two separate processor interfaces. In one, it appears as part of the memory space of a processor and is accessed as though it

were a memory device. A chip select input, $\overline{\mathbf{CHIPSEL}}$, enables the CS8900A when it is used as a memory-mapped device. When it is used as a device within an I/O space, there is no externally generated chip select. Instead, devices mapped into the I/O space of an ISA bus are expected to do their own address decoding, and *that* is why the CS8900A has 20 address lines. Inside the CS8900A is an address decoder specifically for this chip. When the CS8900A is reset, it defaults to I/O address 0x00300. This address can be remapped under software control by writing to the appropriate register of the CS8900A. When used as an I/O-mapped device, $\overline{\mathbf{CHIPSEL}}$ is ignored, and the CS8900A will respond to the appropriate address on its address inputs in conjunction with $\overline{\mathbf{IOR}}$ (I/O read) and $\overline{\mathbf{IOW}}$ (I/O write). You can use the CS8900A in I/O mode within a memory-mapped I/O system. The system address decoder includes the address allocation for the CS8900A but simply does not select it. What the system address decoder must do is ensure that no other device is selected when the address(es) corresponding to the CS8900A are being accessed.

The default setting for the CS8900A is I/O mode operation. To use the CS8900A in memory-mapped mode and therefore to have it recognize $\overline{\mathbf{CHIPSEL}}$ and its memory read ($\overline{\mathbf{MEMR}}$) and memory write ($\overline{\mathbf{MEMW}}$) inputs, the CS8900A must first be accessed as an I/O-mapped device and reconfigured in software. Therefore, to use the memory-mapped option, you still have to support the I/O-mapped addressing scheme to get to it! Therefore, it is much simpler to stick with the I/O-mapped mode and map this within your memory space as just described. If you're using the CS8900A with a processor that has only a 16-bit address bus, simply tie the additional address inputs of the CS8900A to ground. The CS8900A's default address of 0x00300 may be inconvenient for use with some processors that already have internal I/O systems mapped within that region. An access to that address will be intercepted by the internal I/O and never reach the CS8900A. In such cases, the CS8900A's address can't be remapped through software. You will simply never reach the appropriate register. But there is a solution, and it lies within hardware. If you invert some of the address bits from the processor before they reach the CS8900A, you can perform the remapping automatically. The CS8900A still thinks it lies at address 0x00300, but to the processor it is accessed at a completely different address. Figure 11-14 shows an example of this for a processor with a 16-bit address bus.

In this example, address bit **A15** is inverted. So, when the processor accesses address 0x8300 (%1000 0011 0000 0000), this is converted to address 0x0300 (%0000 0011 0000 0000), which is recognized by the CS8900A.

The CS8900A also has support for a serial EEPROM. This can be used to store CS8900A configuration information and the system's unique Ethernet address. Note that this EEPROM is optional, as the host processor can store this data elsewhere in the system. Figure 11-15 shows the CS8900A interfaced to a configuration EEPROM. The interface is standard SPI, and the appropriate pins of the CS8900A are directly connected to the corresponding EEPROM pins. The only other

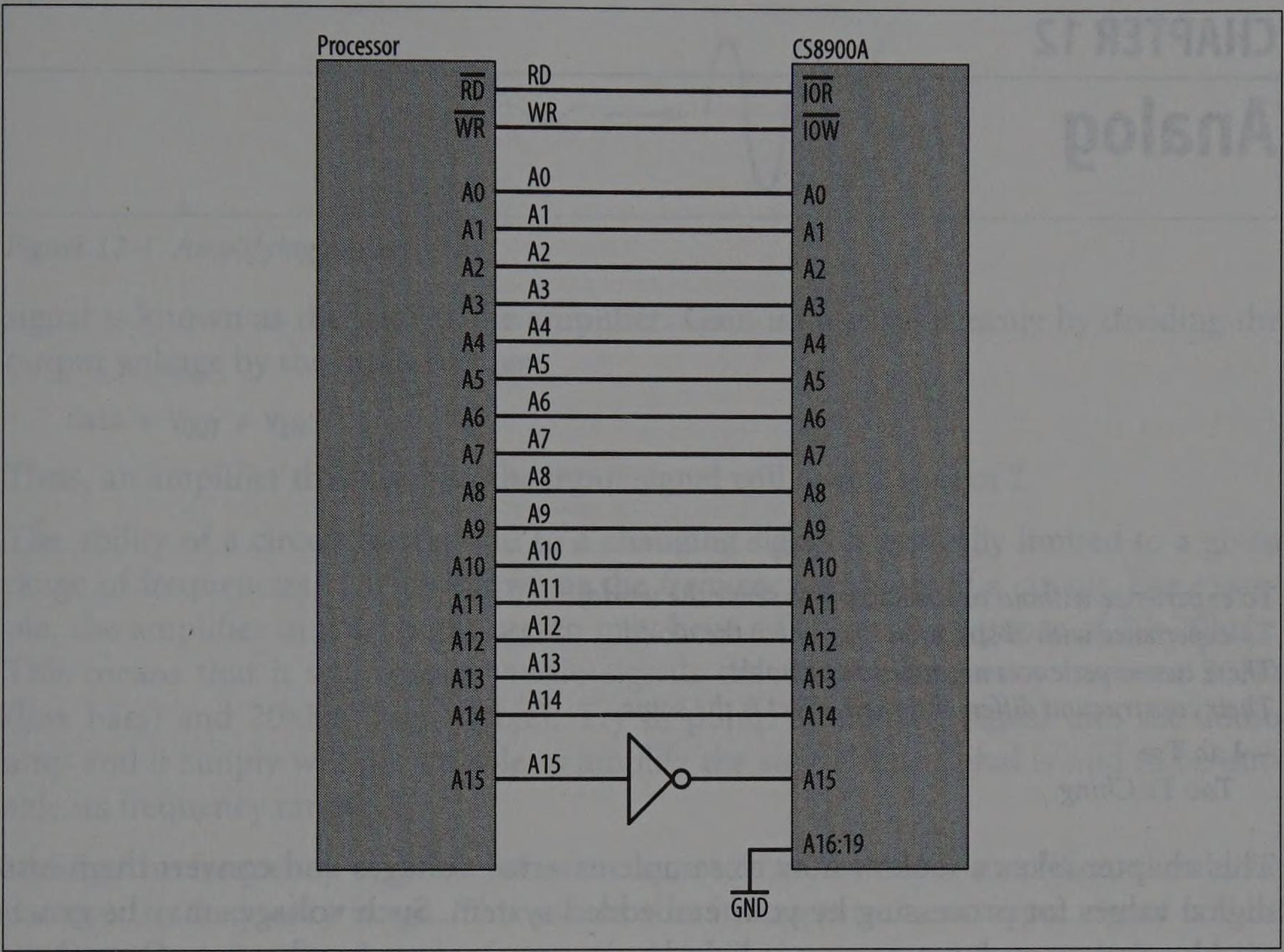


Figure 11-14. Address remapping in hardware

component required is a decoupling capacitor for the EEPROM's power-supply pin. The EEPROM interface is disabled in 8-bit mode, so the host processor must supply all configuration information.

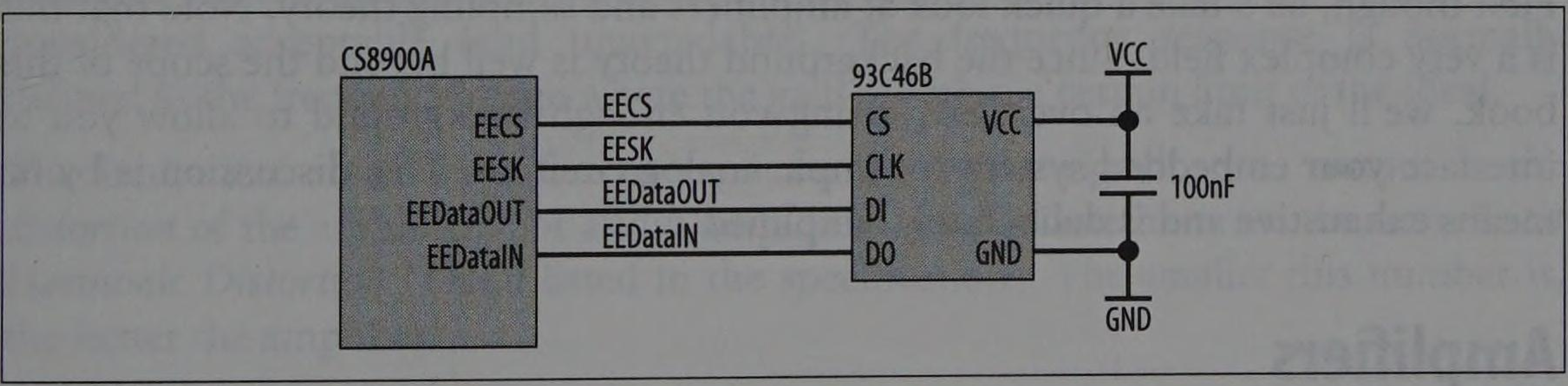


Figure 11-15. CS8900A interfaced to a configuration EEPROM

Finally, any used inputs, such as the DMA signals (**DMACK0**, **DMACK1**, and **DMACK2**), **TEST**, **SLEEP**, **MEMW**, **MEMR**, **AEN**, and **REFRESH**, should be tied inactive. These signals are not used in a typical embedded system.

CHAPTER 12
Analog

To experience without abstraction is to sense the world;
To experience with abstraction is to know the world.
These two experiences are indistinguishable;
Their construction differs but their effect is the same.

—Lao Tse
Tao Te Ching

This chapter takes a look at how to sample external voltages and convert them into digital values for processing by your embedded system. Such voltages may be generated by sensors and may represent light levels, temperature, or vibration. Or perhaps the voltages are the output of a microphone or audio system and need to be converted into digital data. Later, we'll take a look at how you turn digital data into an analog output voltage. The chapter concludes with a look at hardware to control electric motors.

First though, let's take a quick look at amplifiers and sampling theory. Note that this is a very complex field. Since the background theory is well beyond the scope of this book, we'll just take an overview, giving you enough background to allow you to interface your embedded system to simple analog circuitry. This discussion is by no means exhaustive and is deliberately simplified.

Amplifiers

Amplifiers are used to interface one analog circuit to another. An amplifier is a circuit that increases (or decreases) a given input voltage to produce an output voltage. For example, say you had a sensor that produced a maximum output that was 5mVpp, and this was to be interfaced to a sampling system that required an input signal of 5Vpp. You would use an amplifier between the sensor and the sampling system to increase the sensor's output accordingly (Figure 12-1).

The waveform of the amplifier's output signal should be identical to the input signal; only its amplitude will have changed. The amount of increase or decrease in the

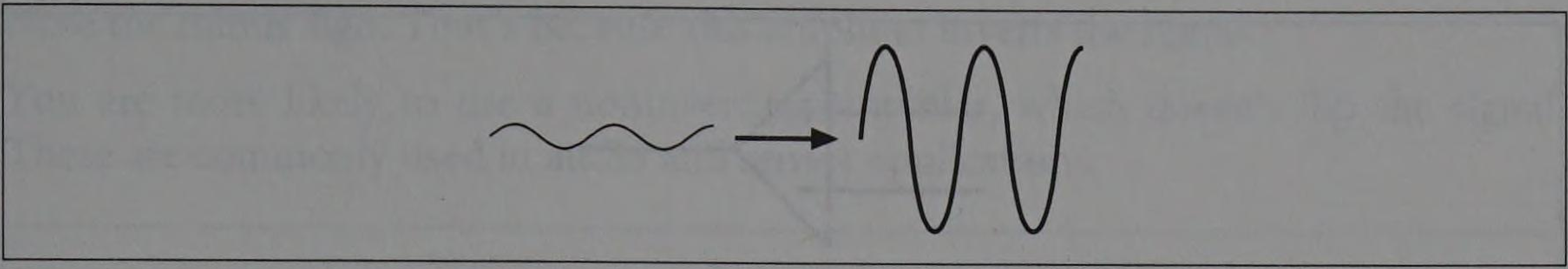


Figure 12-1. Amplifying a waveform

signal is known as the *gain* of the amplifier. Gain is calculated easily by dividing the output voltage by the input voltage:

```
Gain = V_OUT / V_IN
```

Thus, an amplifier that doubles the input signal will have a gain of 2.

The ability of a circuit to respond to a changing signal is typically limited to a given range of frequencies. This is known as the *frequency response* of a circuit. For example, the amplifier in your home stereo may have a frequency response of 20–20kHz. This means that it will amplify audio signals that have a frequency between 20Hz (low bass) and 20kHz (high treble). Try to pump a 100MHz signal into the audio amp and it simply will not be able to amplify the signal. The signal is said to be outside its frequency range.

Ideally, the frequency response of a circuit, such as the audio amplifier, should be flat over its frequency range. This means that its response to an input signal will be the same, no matter what the frequency (within the appropriate range). So, in the case of the audio amp, the gain will be constant for any frequency of signal in the appropriate range. Thus, the volume will not vary with frequency (ignoring any differences due to the original music). At either end of the frequency range, the ability of the amplifier to perform ideally diminishes. At these extremes of frequency, the amplifier's gain diminishes. This is known as *roll off*. Some small degree of roll off is considered acceptable (and unavoidable). The frequency response is normally defined as the frequency range where the gain is within a certain limit of the ideal.

The limitation of an amplifier to replicate the input signal at its output is the *distortion* of the amplifier. For audio amplifiers, you'll sometimes see the term *Total Harmonic Distortion* (*THD*) listed in the specifications. The smaller this number is, the better the amplifier.

In days of old, amplifiers were constructed using *discrete transistors** or *vacuum tubes* (also known as *valves*). These days, amplifiers are available packaged in integrated circuits. These amplifiers are known as *operational amplifiers*, or *op amps* for short. They make the designer's life much easier. They are cheap, reliable, and so very easy to use. Throughout this chapter, whenever we need to amplify a circuit, we'll use an appropriate op amp for the job. The schematic symbol for an op amp is shown in Figure 12-2.

* In some special applications, amplifiers may still be constructed using discrete transistors (or even valves).

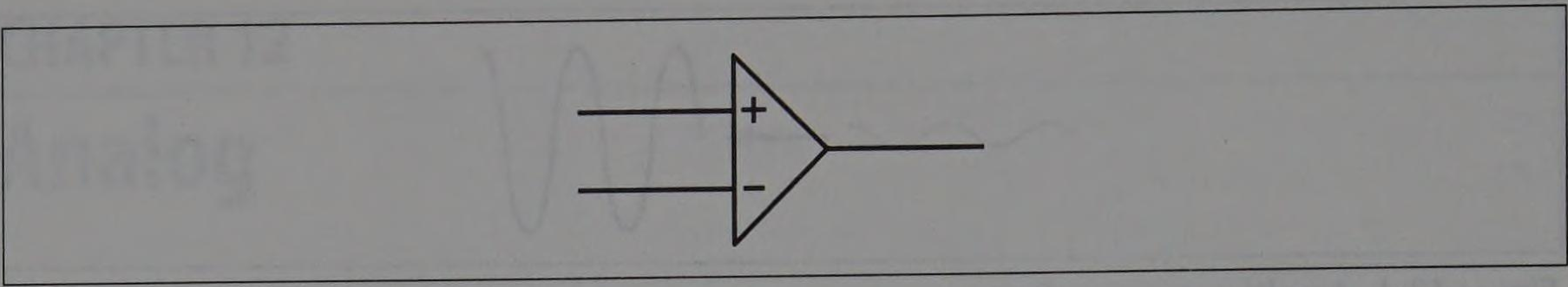


Figure 12-2. Schematic symbol for an op amp

The input marked with + is known as the *noninverting input*, and the input marked with - is the *inverting input*. If the voltage present at the noninverting input is greater than that present at the inverting input, the output of the op amp is positive. Conversely, if the noninverting input is less than the inverting input, the output is negative. Typically, an op amp's output will not go as low as its negative power supply nor as high as its positive power supply, due to the limitations of the internal circuitry. An op amp whose output voltage range does span the difference between its positive and negative power supplies is said to have *rail-to-rail operation*.

In order to function correctly, an op amp requires feedback. Feedback involves coupling the output of an amplifier back to its input. Negative feedback uses the output to reduce the gain of the amplifier and, in doing so, improves the amplifier's other characteristics, such as the flatness of the frequency response and immunity to distortion. Negative feedback is achieved simply by connecting a resistor between the output and the inverting input, as we will shortly see. (A circuit with no feedback is said to be *open loop*.) Op amps are designed so that the outputs change to cancel the difference between the inputs, via a feedback resistor. Thus, the output waveform follows the difference between the input waveforms. The magnitude of the output is proportional to the feedback resistor. The larger the resistor, the more the feedback of the output is attenuated. Thus, the op amp makes the output larger to compensate. In this way, the output is an amplified version of the input.

An op amp may be used as either an inverting amplifier (Figure 12-3) or a noninverting amplifier (Figure 12-4). An inverting amplifier "flips" the signal as well as amplifying it.

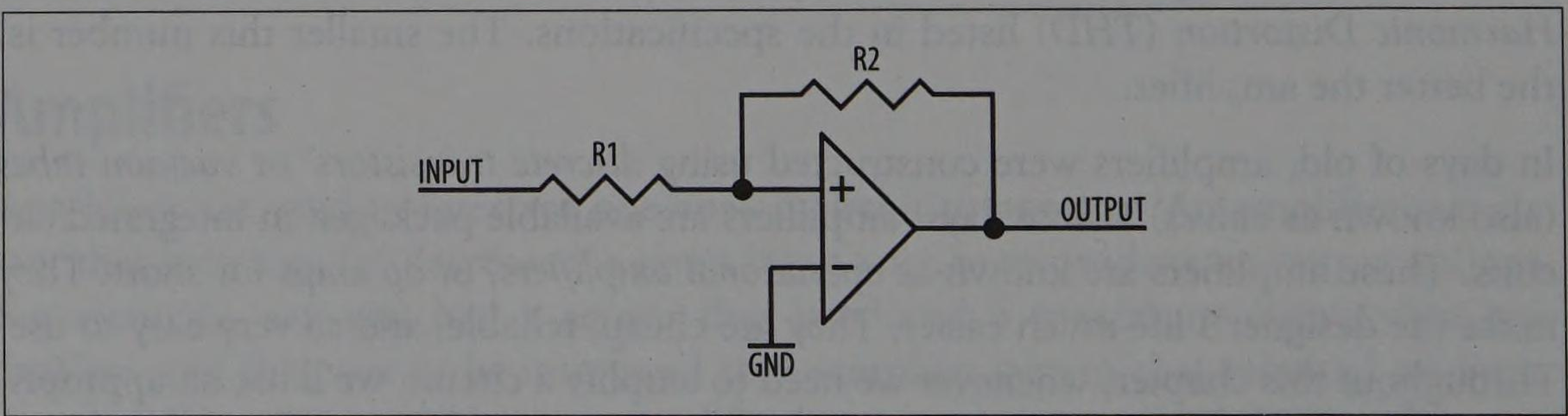


Figure 12-3. Inverting amplifier

The gain of an inverting amplifier is given by:

```
Gain = - R2 / R1
```

Note the minus sign. That's because this amplifier inverts the signal.

You are more likely to use a noninverting amplifier, which doesn't flip the signal. These are commonly used in audio and sensor applications.

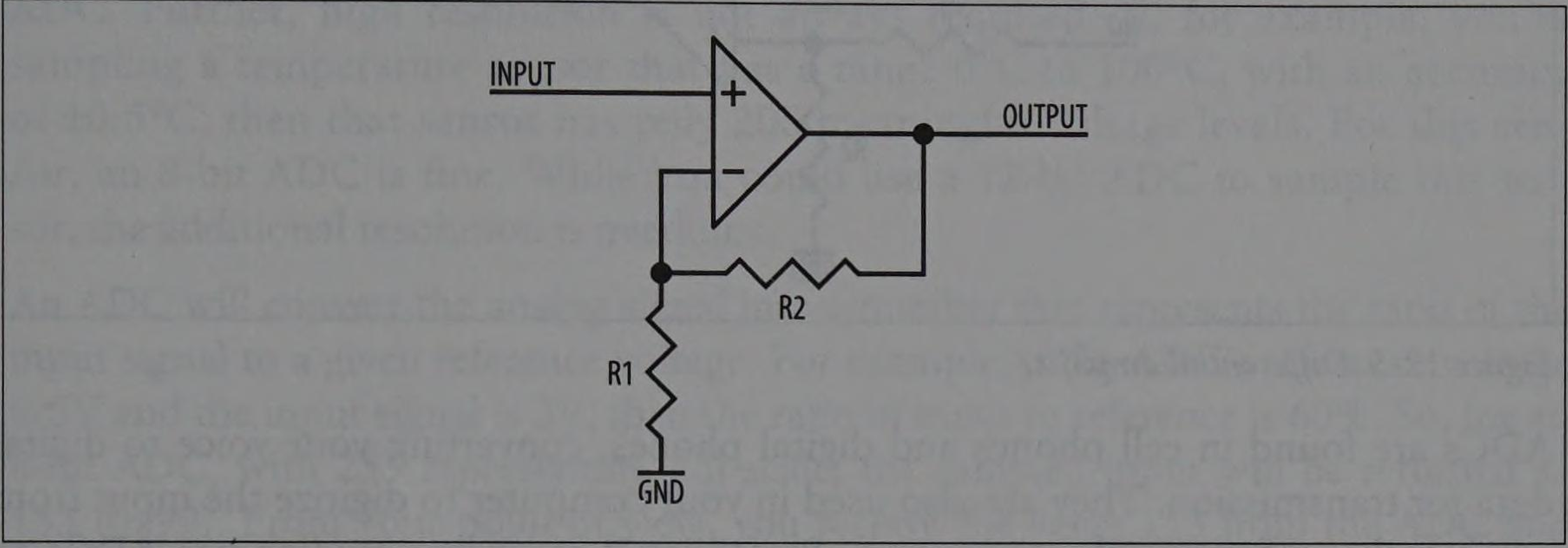


Figure 12-4. Noninverting amplifier

The gain of a noninverting amplifier is given by:

```
Gain = 1 + R2 / R1
```

The gain of the amplifier may be set under software control by using a digital potentiometer (Chapter 9) for R2.

A differential amplifier (Figure 12-5) multiplies the difference between two input signals and is used to amplify small signals that may be subject to noise. By amplifying the difference between the signal of interest and a reference, any noise present is reduced (since the noise will affect both the signal and the reference equally). When both inputs to a differential amplifier change in the same way, this is known as a *common-mode* change. Ideally, a differential amplifier should be immune to common-mode changes, since its purpose is amplifying the signal difference. Its immunity to common-mode changes is known as its *Common-Mode Rejection Ratio* (*CMRR*). The higher the CMRR, the better. To achieve a high CMRR, it is important to match the values (and tolerances) of the resistors as closely as possible.

The output voltage of this differential amplifier is given by:

$$V_{OUT} = (In2 - In1) * (R2 / R1)$$

Analog-to-Digital Conversion

A device that converts an analog input voltage to a digital number is known as an *Analog-to-Digital Converter*, or simply and more commonly as an *ADC*. You may have also heard the term *codec* (*COder-DECoder*) before. A codec is an ADC combined with a *Digital-to-Analog Converter* (*DAC*), providing both analog input and analog output in the one chip. We'll look at DACs in more detail later in this chapter.

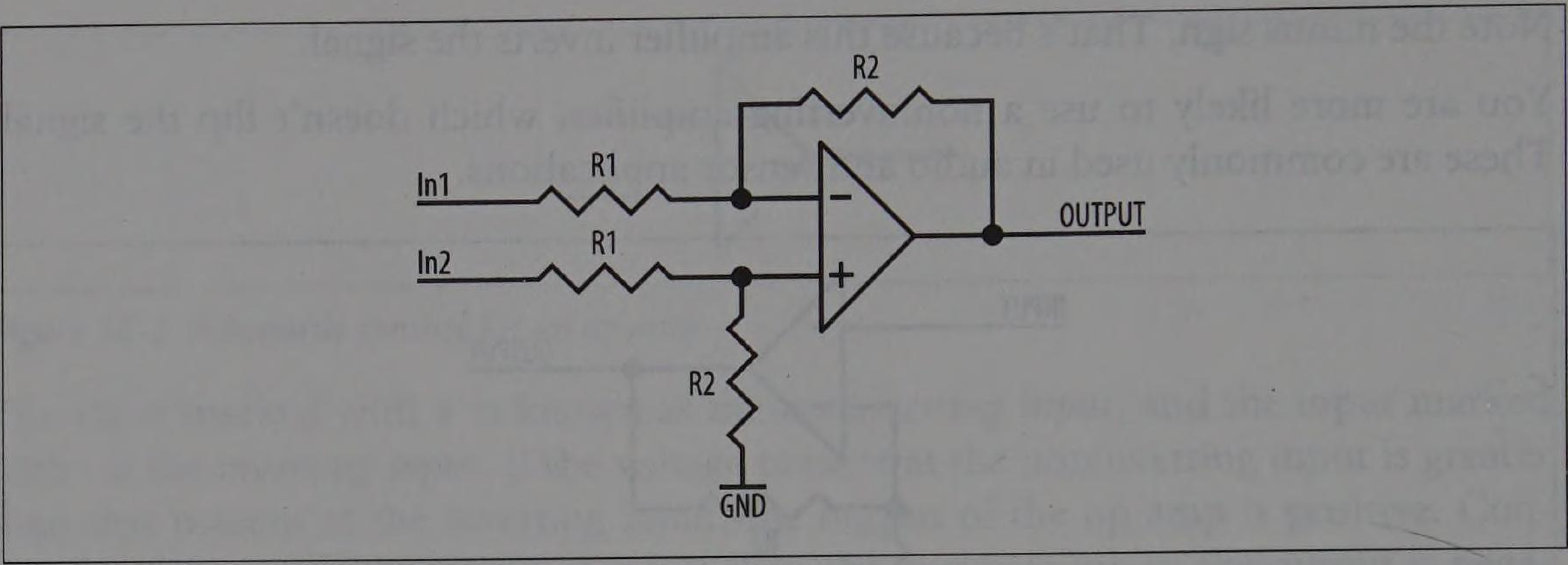


Figure 12-5. Differential amplifier

ADCs are found in cell phones and digital phones, converting your voice to digital data for transmission. They are also used in your computer to digitize the input from a microphone for speech recognition. Professional recording studios use ADCs to convert audio to digital data in preparation for CD mastering. Similarly, video is sampled using ADCs prior to DVD mastering. Your scanner, web cam, and digital camcorder all have ADCs in them. At the other end of the application spectrum, ADCs are used to sample inputs from sensors. These applications can range from automated weather stations to the system monitoring the processor temperature in your PC.

There are several different types of ADCs. *Integrating ADCs* use an internal voltage-controlled oscillator to produce a clock signal whose frequency is proportional to the voltage being sampled. The clock signal is used to drive a counter, which provides the digital value for the sample. The higher the sampled voltage, the higher the clock frequency, and therefore the higher the number reached by the counter. The counter is reset prior to each conversion. Because of this conversion technique, integrating ADCs are not known for their speed of conversion.

A *successive-approximation ADC* uses a DAC to provide an analog reference voltage that is compared to the input voltage. By incrementing the digital code driving the DAC, the reference voltage is increased until a match is found. Once this happens, the code used to drive the DAC is used as the digital output of the ADC.

Flash ADCs (also known as *parallel ADCs*) use a bank of comparators to compare the input voltage with a range of reference voltages. The conversion of the input analog voltage to a digital value is therefore very fast. The catch is that flash ADCs tend to be more expensive than other types of ADCs and due to their complexity normally have a lower resolution than other forms of ADCs.

The process of converting an analog signal to digital is known as *sampling* or *quantization*. ADCs have two principal characteristics—sample rate and resolution. Sample rate is expressed as samples per second (SPS) and refers to how frequently an analog input signal is converted into a digital code. The faster an ADC's sample rate, the more expensive that chip will be. Resolution determines the accuracy of each

sample. For example, an 8-bit ADC will return an 8-bit code representing the sampled input signal. This means that the input has been quantized into one of 256 discrete values. A 12-bit ADC will quantize the signal into one of 4096 values, yielding a more accurate result. However, the higher the resolution, the more expensive the ADC. Further, high resolution is not always required. If, for example, you're sampling a temperature sensor that has a range 0°C to 100°C, with an accuracy of ±0.5°C, then that sensor has only 200 meaningful voltage levels. For this sensor, an 8-bit ADC is fine. While you could use a 12-bit ADC to sample this sensor, the additional resolution is overkill.

An ADC will convert the analog signal into a number that represents the ratio of the input signal to a given reference voltage. For example, if the ADC's reference voltage is 5V and the input signal is 3V, then the ratio of input to reference is 60%. So, for an 8-bit ADC, with 255 representing full scale, the sampled input will be returned as 153 (0x99). From your point of view, you receive the value 153 from the ADC and from this must work back to calculate the original analog voltage.

```
Signal = (sample / max_value) * reference_voltage
       = (153 / 255) * 5
       = 3 Volts
```

Sample Rates

The rate at which a signal is sampled can have a dramatic effect on the quantized result and therefore can also affect the way in which software interprets that result. Figure 12-6 shows a sinusoidal signal that is sampled at a rate equal to its period. In this example, the sample happens to coincide with a peak in the signal. The signal changes in between samples, but our choice of sample rate means that we get the same value each time. We get a completely false picture of what is really happening to that signal. To our sampling software, each value returned is the same, and so the signal appears to us as though it were a flat line!

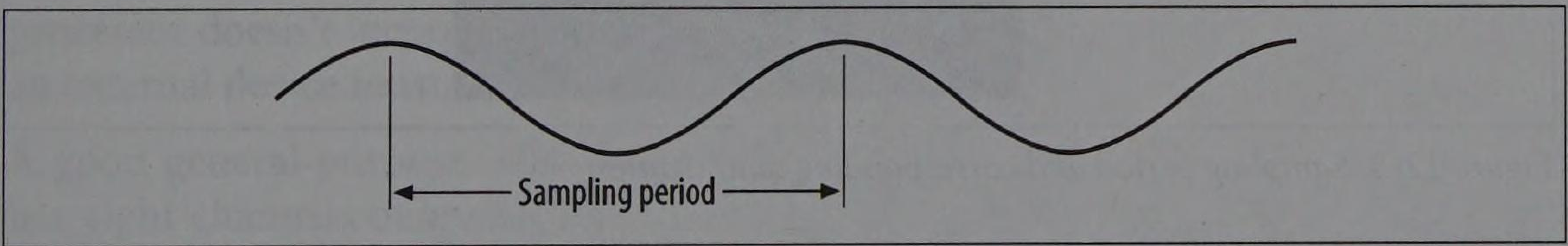


Figure 12-6. Poorly chosen sample rate gives inaccurate signal reading

If we choose a sample rate that is double (or more) the signal's highest frequency component, we can see the signal in more detail (Figure 12-7). This sampling frequency is known as the *Nyquist frequency* and is the lower limit of what will produce usable results. If the sample rate is slower than the Nyquist frequency, false artifacts (such as our sine wave appearing as a straight line, as we saw previously) may appear in the sampled result. These phantoms are known as *aliasing*.

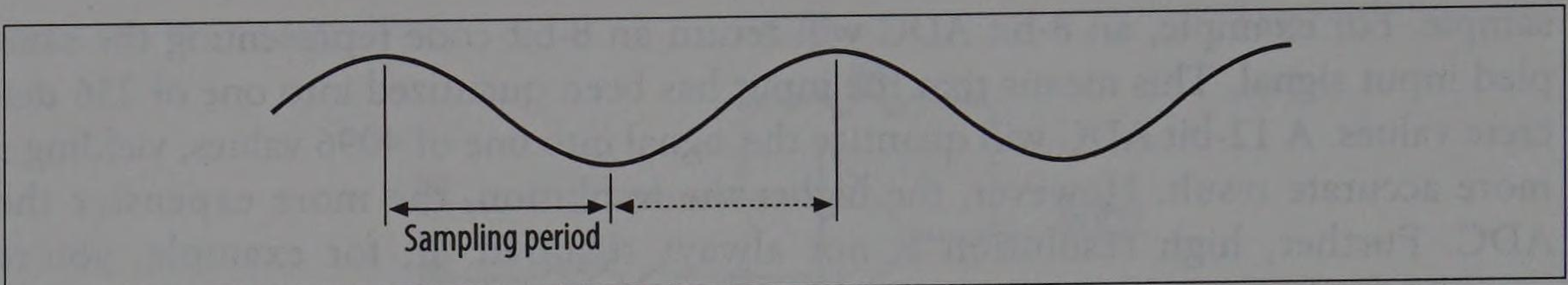


Figure 12-7. Shorter sampling period

> Ever see an old western movie when the wheels of a wagon appear to be rotating backward, even though the wagon is moving forward? That's an example of aliasing. The frame rate of the camera is effectively sampling the rotation of the wheels. Because the wheel rotation is slightly slower than the frame rate, the wheel doesn't quite make a full revolution per frame. So on each successive frame, the wheel appears a little further behind than it was on the preceding frame. The effect is as though the wheel is rotating backward—aliasing in action!

The faster the sample rate, the more accurate your sampled results will be. Since your sampling is quantizing the signal both in terms of amplitude (ADC resolution) and time (sample rate), a quantization error will always result (Figure 12-8).

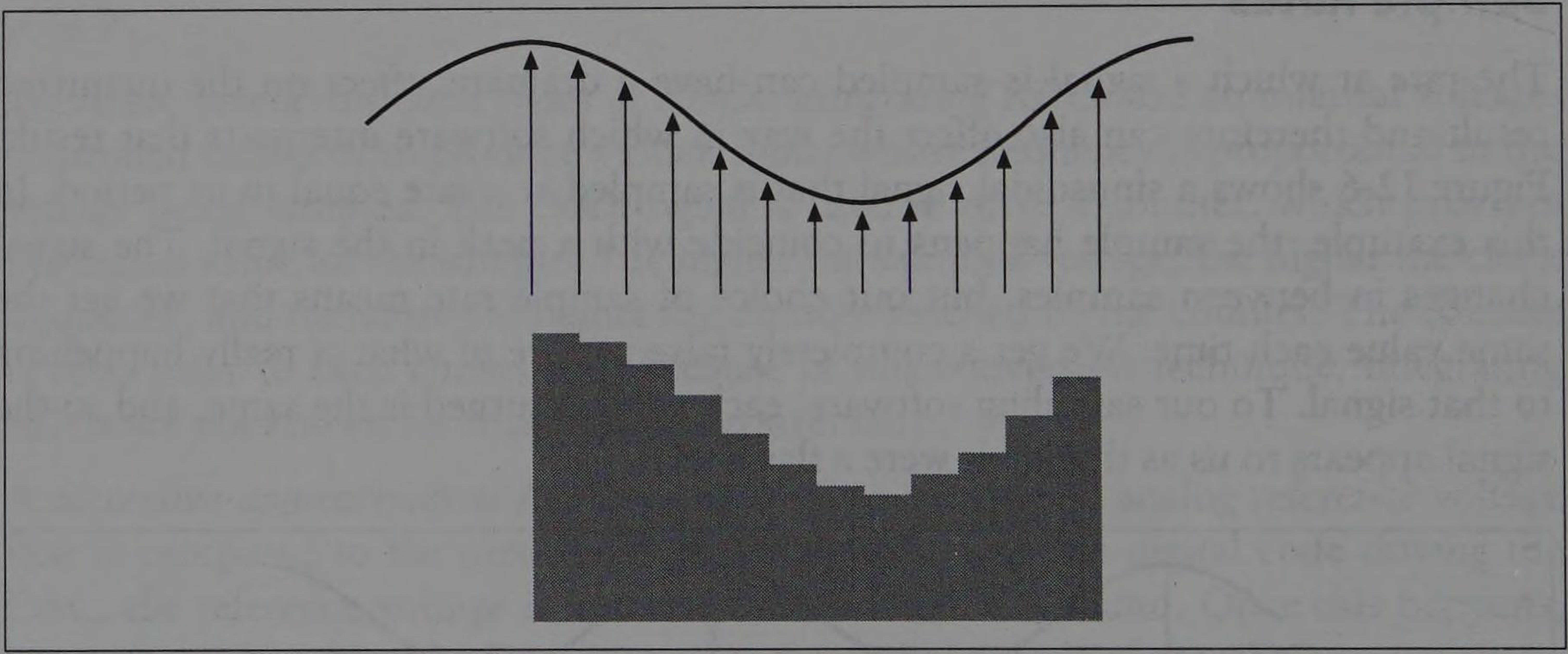


Figure 12-8. Sampling period and corresponding quantization

As you can see, the smooth sine wave of the original signal has become a jagged representation. Now, if you are monitoring temperature, this may be sufficient. You may not care how the temperature signal changed. Instead, you may be interested in the temperature only at specific intervals and with only limited accuracy. In which case, this effect is not really a problem.

However, if you are sampling audio, this quantization effect can be a real problem. By increasing the sample rate, a more accurate representation of the original signal is obtained (Figure 12-9).

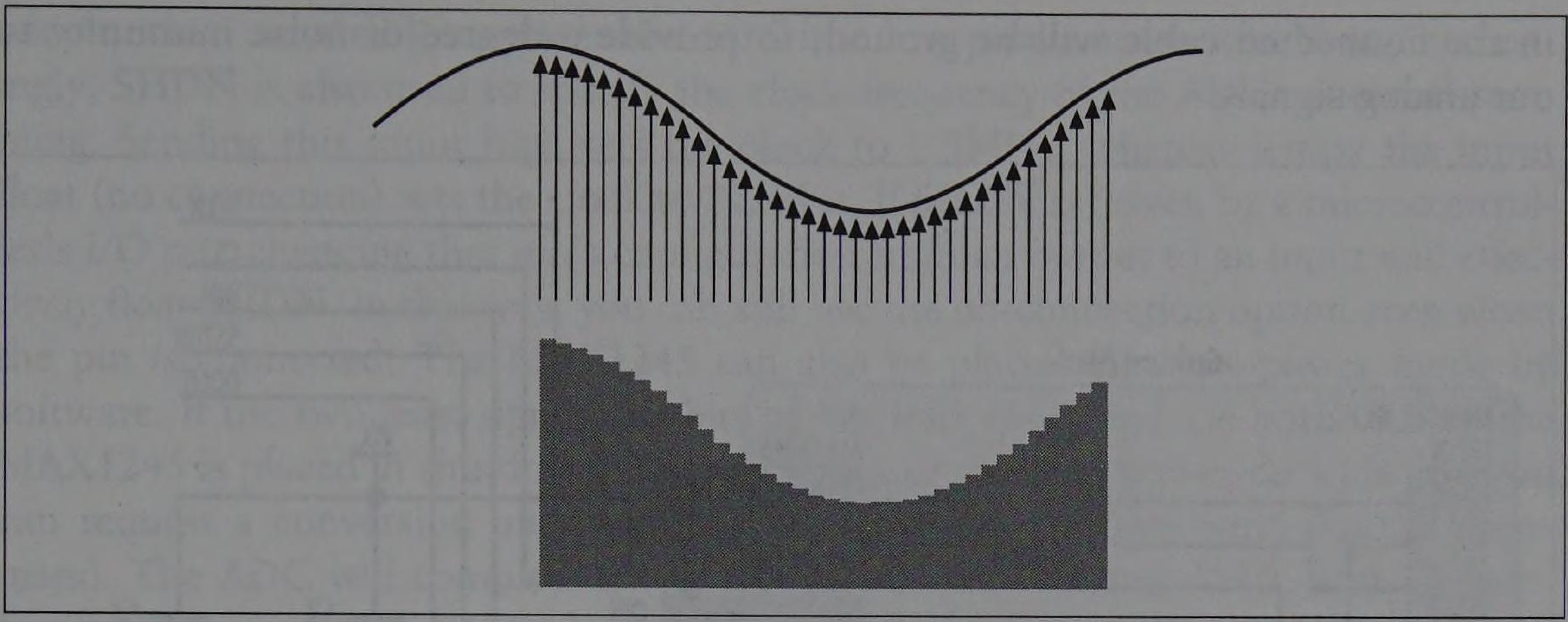


Figure 12-9. Fast sample period results in less quantization

A voice mail system may use a sample rate of only 8kHz and a resolution of 12 bits, and the resultant sound quality is limited. However, CD audio uses a sample rate of 44.1kHz with 16-bit data and achieves a significant improvement in quality as a result. DVD audio uses a sample rate of 48kHz with 24-bit data for even greater audio fidelity. To further improve sound quality, both CD and DVD players have special output filters to smooth the transitions between each sample when the data is converted back into analog form.

The take-home message is: choose your ADC resolution and sample rate carefully, keeping in mind exactly what you're sampling and what you intend to use it for.

Interfacing an External ADC

A very wide range of ADCs is available, for every considerable purpose. Choose from very low-cost, low-speed ADCs for simple voltage conversion to very high-speed, precise (and expensive) ADCs for sampling video streams. Many microcontrollers have built-in ADC subsystems, making analog interfacing simple. However, if the processor doesn't incorporate an ADC, or its ADC is not suited to your application, an external device must be added.

A good general-purpose ADC for sensor applications is the Maxim MAX1245. It has eight channels of analog input and can sample at 100,000 samples per second, with a resolution of 12 bits. (Similar devices have resolutions ranging from 8 bits to 16 bits, with interfaces such as SPI, I²C, and processor bus.) The MAX1245 has an internal track and hold, preventing a changing signal from corrupting the result during a conversion. The MAX1245 is interfaced to a host processor via an interface that is compatible with SPI, Microwire, and the serial interfaces found in Texas Instruments TMS320-series DSP processors (Figure 12-10). As you can see, the MAX1245 is very easy to use. In this schematic, the analog input is coming in via an IDC header, the 16-pin connector on the left of the figure. Note that every second pin on the connector is tied to ground. This means that every second wire

in the connected cable will be ground, to provide a degree of noise immunity to our analog signals.

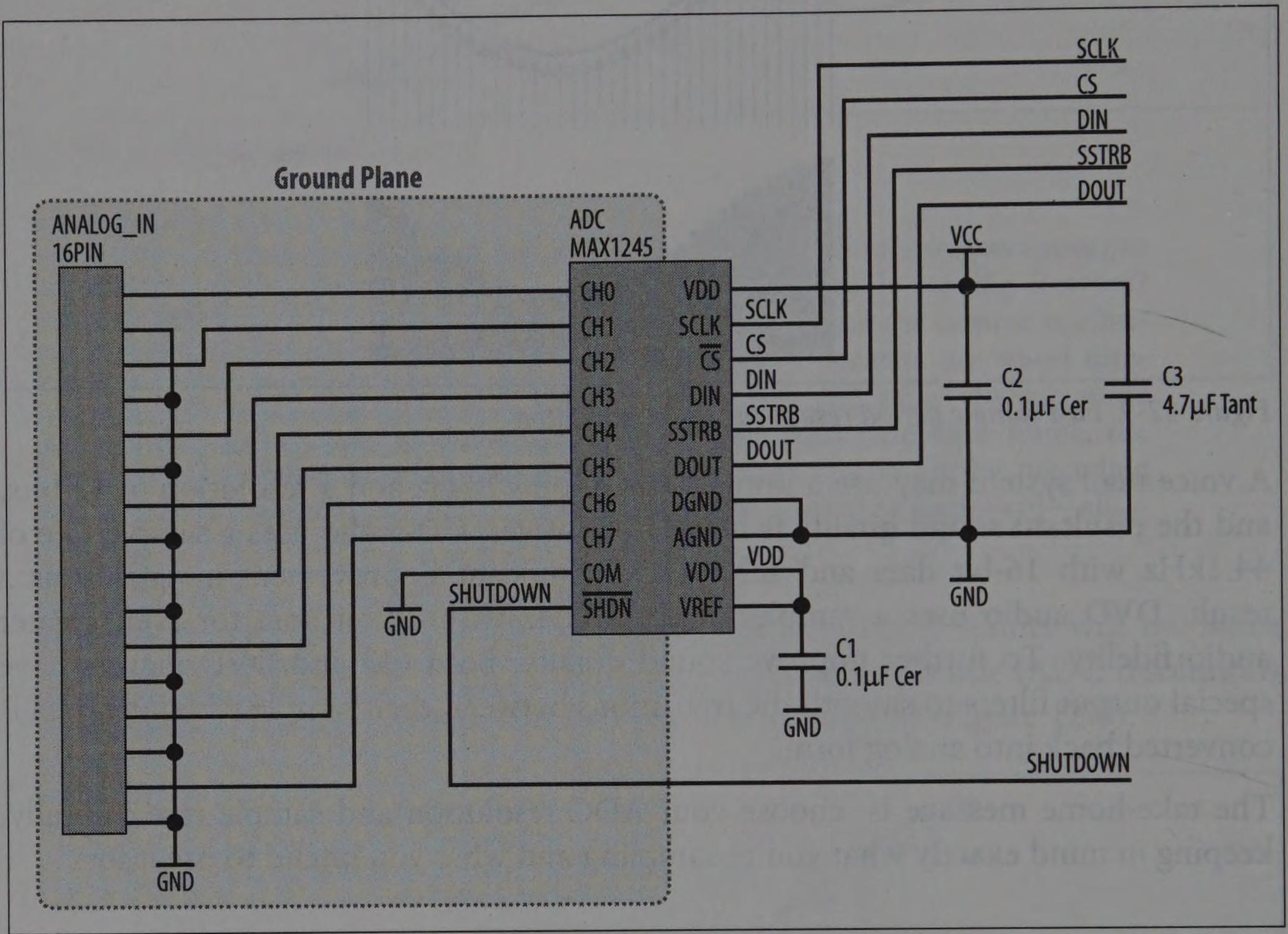


Figure 12-10. MAX1245 interface

The **DOUT**, **DI**, and **SCLK** signals correspond to a processor's SPI **MISO**, **MOSI**, and **SCLK** signals, respectively. $\overline{\text{CS}}$ is simply generated using a processor I/O line.

A conversion commences by sending a start command to the ADC via the SPI interface. The start command is simply a byte that specifies the channel and other ADC settings for that particular conversion. (Refer to the MAX1245 datasheet for more information on the software interface.) The MAX1245 may either use an internal clock source to drive the conversion process or have an external clock provided. The SPI **SCLK** also doubles as the conversion clock, when the ADC is used in external clock mode. When used in internal clock mode, the output, **SSTRB** (Serial Strobe), goes low at the beginning of a conversion and returns high once the conversion is complete. When an external clock is used, **SSTRB** pulses high in the clock period prior to the most significant bit being processed. **SSTRB** may be used to flag the completion of a conversion to a host processor, by acting as an interrupt input. Alternatively, when used in external clock mode, the conversion result is ready once the start command has been sent.

The MAX1245 has the ability to enter low-power mode. This can be done either through hardware or software control. The MAX1245 has an input pin, $\overline{\textbf{SHDN}}$,

which, when asserted low, places the ADC into low-power operation. Now, interestingly, $\overline{\textbf{SHDN}}$ is also used to specify the clock frequency of the ADC's internal sampling. Sending this input high sets the clock to 1.5MHz, whereas letting the input float (no connection) sets the clock to 225kHz. If $\overline{\textbf{SHDN}}$ is driven by a microcontroller's I/O pin, changing that pin's configuration from an output to an input will effectively float $\overline{\textbf{SHDN}}$. In this way, you can still use the no-connection option even when the pin *is* connected! The MAX1245 can also be placed into low-power mode by software. If the two least-significant bits of the start command are both 0, then the MAX1245 is placed in shutdown. The advantage of software power-down is that you can request a conversion *and* place the device into shutdown with a single command. The ADC will complete the conversion before shutting down, and its interface will remain active so that the result may be clocked out to the microcontroller.

Power for the MAX1245 (**VDD**) can be in the range 2.7V to 3.3V. The MAX1245 has three ground pins, **COM**, **DGND**, and **AGND**. **COM** is the ground reference for the analog inputs, **DGND** is the ground connection for the digital section of the ADC, and **AGND** is the ground connection for the analog section of the ADC. These three grounds need to be connected together, but only at a single point, close to **AGND**. This is known as a *star ground point*. The two power inputs (**VDD**) need *two* decoupling capacitors to remove noise from the supply voltage. A 0.1µF capacitor and a 4.7µF capacitor should be used to decouple **VDD** and should be placed as close to the star ground point as possible. For particularly noisy power supplies, a 10Ω resistor should be placed in series between the power source and **VDD**. The analog inputs should be shielded from all nearby digital signals to prevent interference, and a ground shield (a fill) should be placed under the MAX1245 to further isolate the device from noise. (See Chapter 4 for more information on noise and shielding.)

Now that we have seen how to add an ADC to a microcontroller, let's give it something to sample. We'll now take a look at some sensors and see how to interface them to an ADC. There are lots of different sensors available, from many manufacturers. Many are hard to use, are awkward to interface, and require much more effort than seems necessary. But not all sensors are created equal. I have sought out and selected a range of sensors that are trivial to use and require little or no design effort. Electronics can be hard, but it doesn't always have to be so, as you will see.

Temperature Sensor

We'll start with something simple, a temperature sensor. This little sensor has a wide range of applications. The most obvious is an environment monitor or weather station, but you could also use it to sense temperatures inside rooms and to control the appropriate heating or cooling systems. Combine it with a datalogger design, and you have a temperature recorder. Such devices are used in the shipment of fruits, vegetables, frozen foods, and flowers, to ensure that they get to market in

their best condition. It can also be used in the shipment of blood products and pathology samples, making sure that these critical substances are not exposed to adverse temperatures.

The AD22100 and AD22103 temperature sensors, by Analog Devices, are very easy to use. They are three-pin devices,* requiring only power (V_S) and ground to give you a voltage output that is proportional to temperature (Figure 12-11). The AD22100 requires a 5V supply, and the AD22103 requires a 3.3V supply.

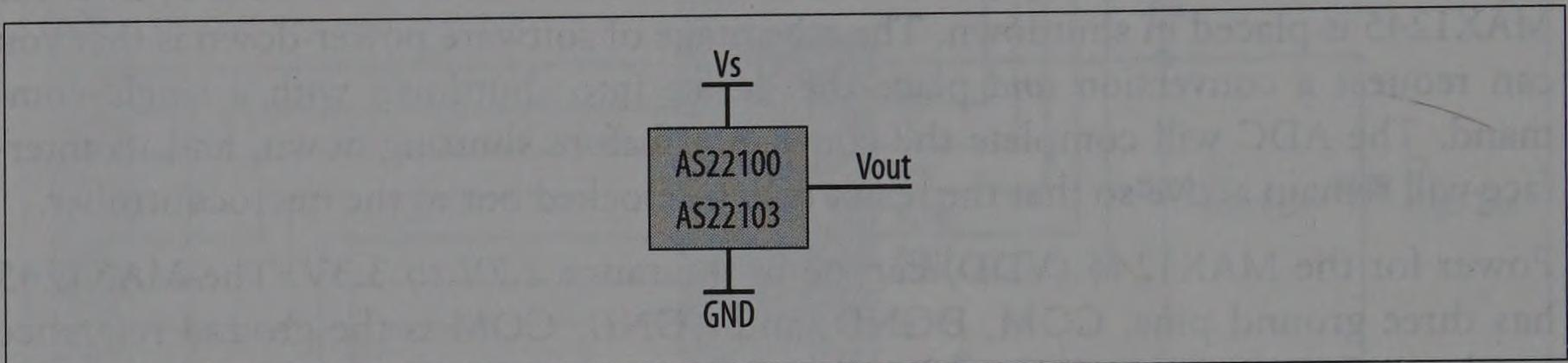


Figure 12-11. AD22100/AD22103

What could be easier than that?

The output voltage corresponds to 22.5mV/°C over the temperature range -50°C to +150°C for the AD22100, and 28mV/°C over the temperature range 0°C to 100°C for the AD22103. The transfer functions (how the output relates to the input) for the two devices are given by:

$$V_{out} = (V_S / 5) \times [1.375 + (0.0225 \times T_A)] \qquad \text{AD22100}$$
$$V_{out} = (V_S / 3.3) \times [0.25 + (0.028 \times T_A)] \qquad \text{AD22103}$$

where $\mathbf{V_{out}}$ is the output voltage, $\mathbf{V_s}$ is the power supply, and $\mathbf{T_A}$ is the ambient temperature.

So, turning the equations around, the relationship between temperature and output voltage is:

$$T_A = (((Vout \times 5) / V_S) - 1.375) / 0.0225 \qquad \text{AD22100}$$
$$T_A = (((Vout \times 3.3) / V_S) - 0.25) / 0.028 \qquad \text{AD22103}$$

For example, if we were using an AD22100 temperature sensor with a supply voltage of 5V ($\mathbf{V_S} = 5V$), then our function becomes simply:

$$T_A = (Vout - 1.375) / 0.0225$$

Thus, if we measured an output voltage of 1.94V, the corresponding temperature would be 25.1°C.

Interfacing the temperature sensor to an ADC is simple. The output may be directly connected to an input of the ADC. Alternatively, since temperature changes relatively slowly, we can add an RC filter between the sensor and the ADC to remove any noise that may be present in the output (Figure 12-12).

* These devices are also available in eight-pin surface-mount chips, where five of the pins are unconnected.

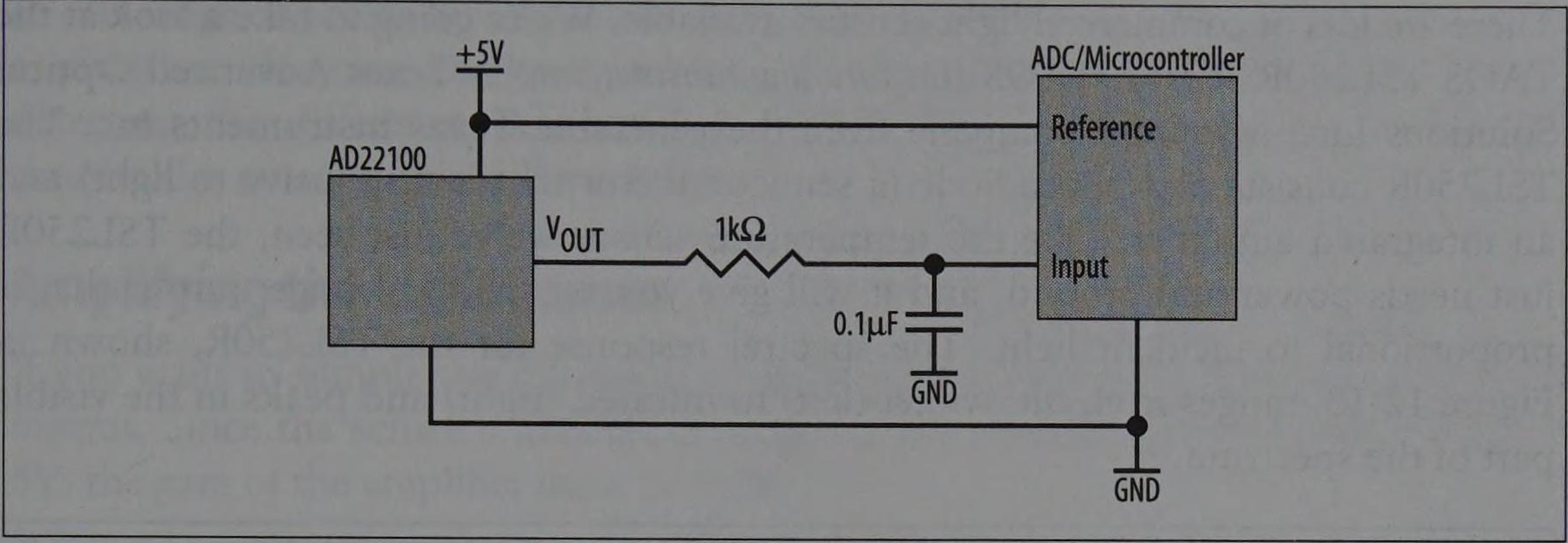


Figure 12-12. ADD22100/AD22103 with an RC filter

Light Sensor

Now we'll take a look at a light sensor. The obvious use is to monitor natural light levels and perhaps use the results to control artificial lighting systems. But, combine this sensor with a directional light source (such as a bright LED enclosed in a baffle), and you have a security detector. As long as the sensor can "see" the LED, everything's fine. But when the light is interrupted, you know that someone or something has passed between.

My company uses the particular sensor we're going to look at on a small datalogger. One of our customers is a biologist who studies albatrosses (giant seabirds) of the southern oceans, as part of an ongoing conservation program. These birds will fly for years at a time, circumnavigating the world on the ocean winds. The tiny datalogger (smaller than your smallest finger) weighs only a few grams and is attached to the bird's leg. (The attachment is designed and fitted with great care to ensure that the bird is not harmed or adversely affected in any way.) The light sensor is used to record sunlight levels that the bird experiences on its journey.

By comparing the recorded sunrises and sunsets with the reference clock aboard the datalogger and looking at the duration of twilight, latitude and longitude can be computed. In this way, the simple recording of light levels is used to track an albatross's journey as it circumnavigates the world.

The recorded light profiles also give information about what the albatross does. You can tell whether the bird was flying with feet tucked up in the feathers, flying with feet hanging down, or resting on the water, as each activity has a unique light profile associated with it. You can also see the phases of the moon leaving their trace on the nighttime light levels, as well as which days were cloudy and which were sunny. It even detects when the albatross stumbles across a lonely, and brightly lit, squid boat during the night. One simple sensor can tell you an awful lot.

There are lots of commercial light sensors available. We're going to take a look at the TAOS TSL250R sensor. TAOS (*http://www.taosinc.com*) is Texas Advanced Optical Solutions Inc., a spin-off company from the venerable Texas Instruments Inc. The TSL250R consists of a photodiode (a semiconductor that is responsive to light) and an integrated amplifier. Like the temperature sensor we've just seen, the TSL250R just needs power and ground, and it will give you an analog voltage output that is proportional to incident light. The spectral response for the TSL250R, shown in Figure 12-13, ranges from ultraviolet (left) to infrared (right) and peaks in the visible part of the spectrum.

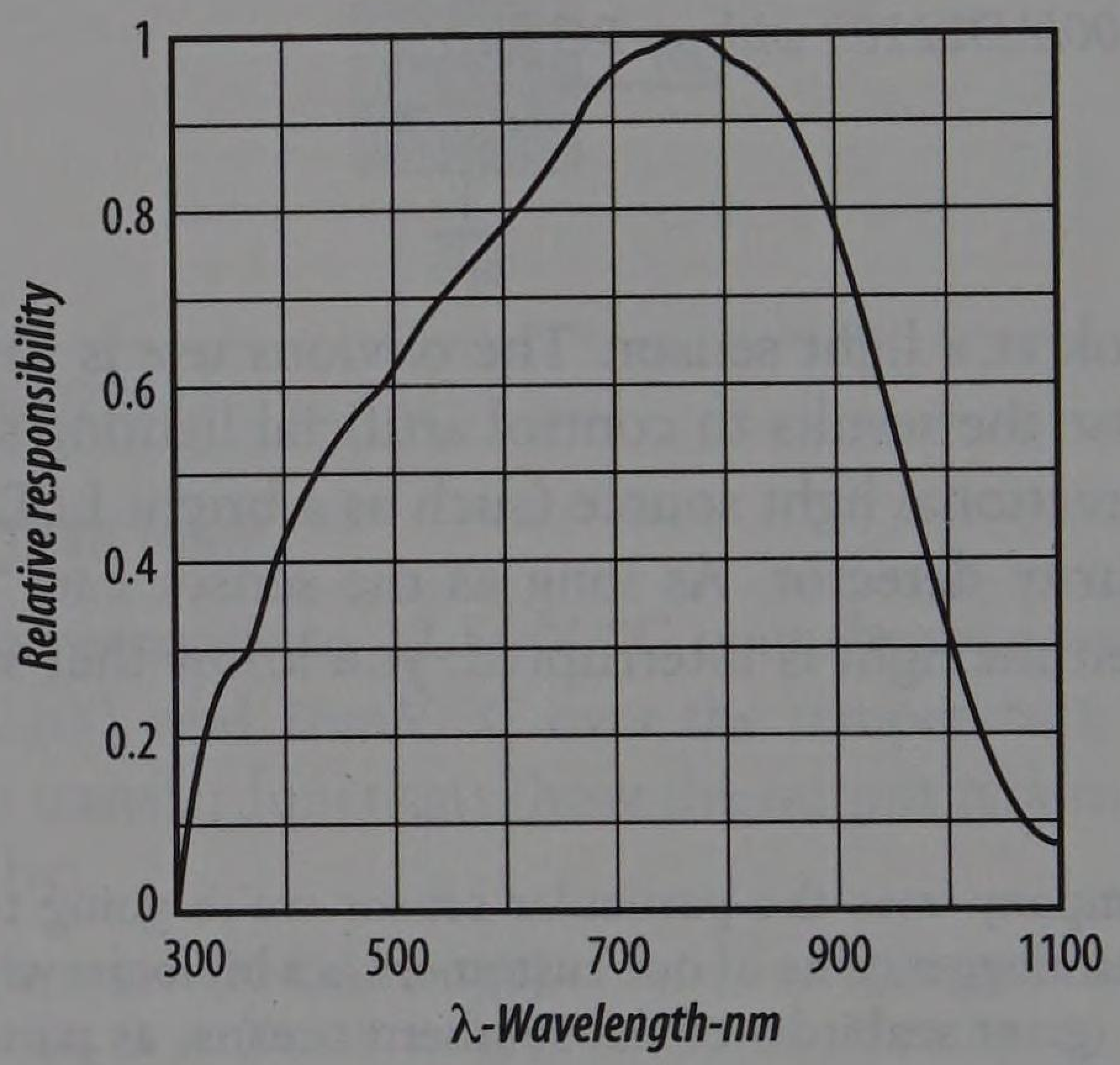


Figure 12-13. Spectral response of a TAOS TSL250R

The TSL250R can operate from a supply voltage of between 2.7V and 5.5V and typically consumes only 1.1mA of current. The basic circuit for the TSL250R is very simple (Figure 12-14).

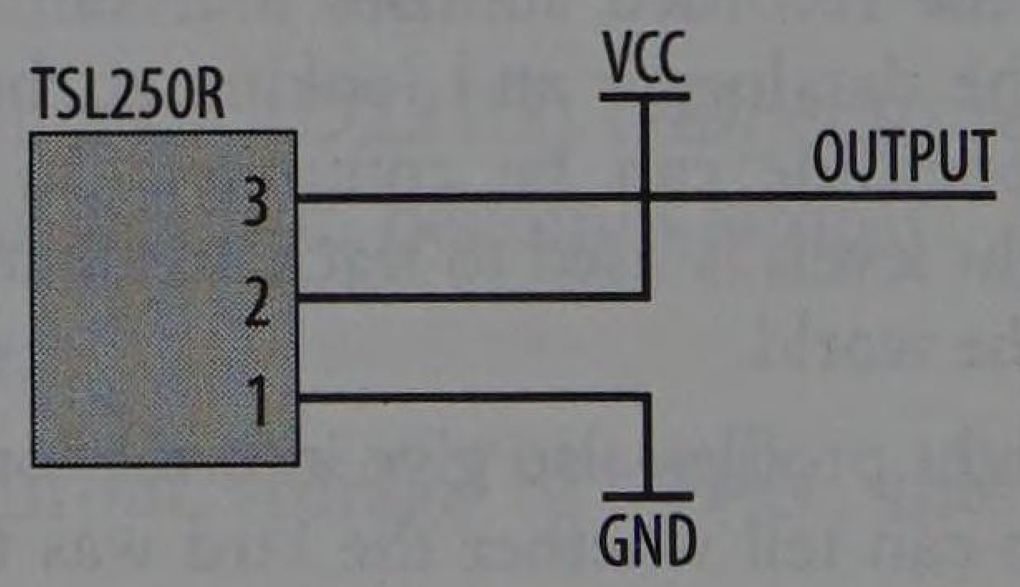


Figure 12-14. Using the TAOS TSL250R

The maximum output voltage (under full irradiance) for this sensor is just under 4V, when the part is powered from a 5V supply. So, if we choose, we can interface this sensor directly to a (5V-referenced) ADC, without any additional amplification. Now, because the output does not span the full scale of the ADC's range, we lose a small amount of resolution. For an 8-bit ADC, a 4V input corresponds to 0xCC, and

so our range of values for this sensor goes from 0x00 to 0xCC. Depending upon your application, this may not be a problem. For example, if you are interested only in detecting the difference between light and darkness or when a given low-light threshold is crossed, this will work fine.

Amplifying the Light Sensor

If you want to sample the full range of the sensor, you need to amplify the sensor's output. Since the sensor's maximum output is 4V and the reference of the ADC is 5V, the gain of the amplifier must be 1.25.

A good general-purpose op amp is the AD623 by Analog Devices. It has rail-to-rail operation, can run from a single supply voltage, requires very little current, and is exceptionally easy to use. Analog Devices has done a lot of the hard work already, and the AD623 requires only a single external resistor to set the gain. The value of the resistor is calculated using the relation:

$$R_G = 100k\Omega \ / \ (\text{Gain} - 1)$$

So, for our required gain of 1.25, we need a resistance of:

$$\begin{aligned} R_G &= 100k\Omega \ / \ (1.25 - 1) \\ &= 100k\Omega \ / \ 0.25 \\ &= 400k\Omega \end{aligned}$$

The resistor should have a tolerance (accuracy) of 1% or better. Standard off-the-shelf resistors are normally 5% and just aren't accurate enough.

The circuit with the TSL250R interfaced to the AD623 is shown in Figure 12-15.

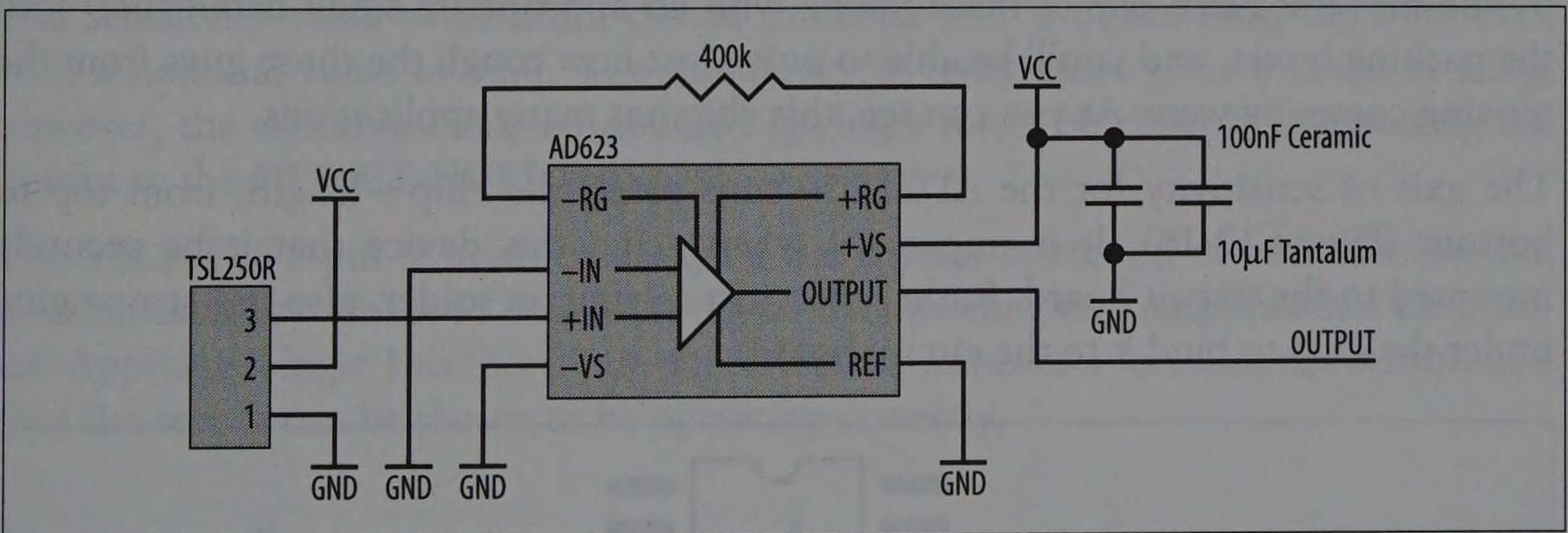


Figure 12-15. Amplifying the output of the TSL250R light sensor

The output of the TSL250R sensor (pin 3) is connected to the noninverting input of the AD623 op amp (pin 3), while the inverting input is tied to ground. The gain resistor is connected between pins 1 and 8. The negative power supply, **-VS**, is connected to ground for single-supply operation. The positive power supply, **+VS**, is connected to **VCC** and is decoupled to ground using two capacitors. The op amp's reference input (**REF**) is also tied to ground. The output of the op amp at pin 6 is then connected directly to the analog input of an ADC.

Accelerometer

Now I'm going to take a look at an interesting sensor. Analog Devices makes some really nice accelerometers, and I'm going to show you how to interface an ADXL150 to an embedded system. You can use an accelerometer for a number of applications, not just measuring linear acceleration of vehicles. The ADXL150 is a single-axis (one-dimensional) accelerometer with a resolution of 10mg and a full-scale range of ±50g. For dual-axis (two-dimensional) sensing, choose the ADXL250.

> g is the unit of acceleration. One g is approximately equal to 9.8 m/sec^2 (32.2 feet/second^2). As a passenger in a commercial jet aircraft, you'll experience a force of about 2g when the aircraft turns. A fighter aircraft will experience a force of around 8g when turning sharply. Without a special suit, the jet fighter pilot would black out under a force of 8g. So, the ADXL150, with a range of ±50g, can measure a significant amount of force!

Such a fine resolution means that you can use this sensor to measure gentle vibrations and shifts. You could use it in a seismometer for geophysical applications or to measure vibrations or ground shift in mines, in tunnels, or at building sites. You could use it to monitor motion and, by placing three accelerometers orthogonally get an accurate 3-D motion recorder. The same setup could also be used as a digital carpenter's spirit level by sensing the direction of the Earth's gravitational field. Perhaps you might use it to monitor violent physical shock, such as crash-test measurements. Ever moved house only to discover that Granny's fine crystal glassware was smashed by the movers? Place one of these (along with an appropriate small datalogger) into the packing boxes, and you'll be able to prove just how rough the those guys from the moving company were. As you can see, this chip has many applications.

The axis of sensitivity for the ADXL150 runs along the chip's length, from top to bottom (Figure 12-16). It is important when using this device that it be securely mounted to the circuit board. Rather than just relying on solder, also use strong glue under the chip to bind it to the circuit board.

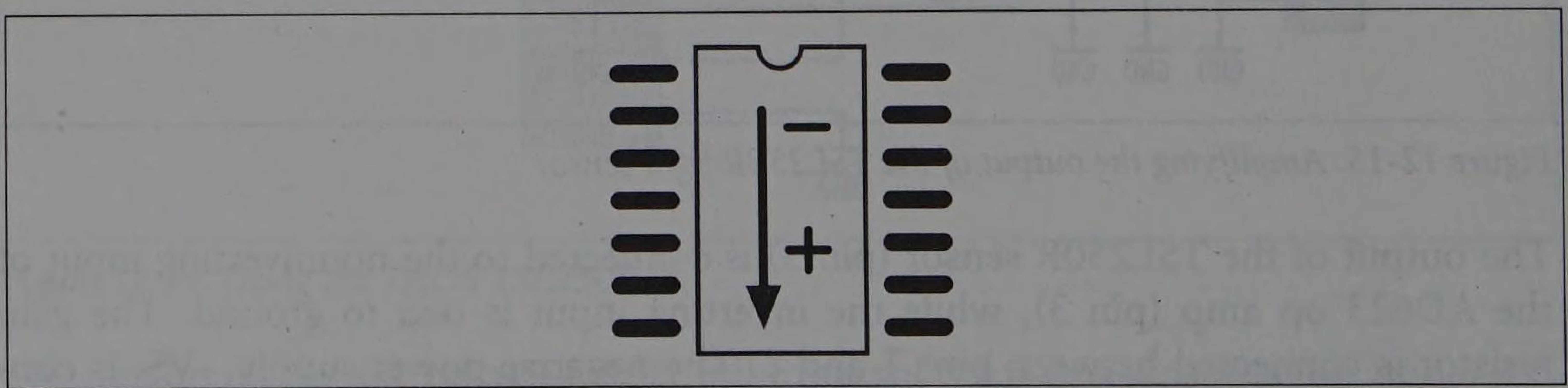


Figure 12-16. Axis of sensitivity

The ADXL150 requires no external components (save for power-supply decoupling) and is a completely self-contained unit, incorporating not only the sensor but also

signal conditioning and amplification. Its output can be interfaced directly to an ADC. The schematic for using the ADXL150 is shown in Figure 12-17.

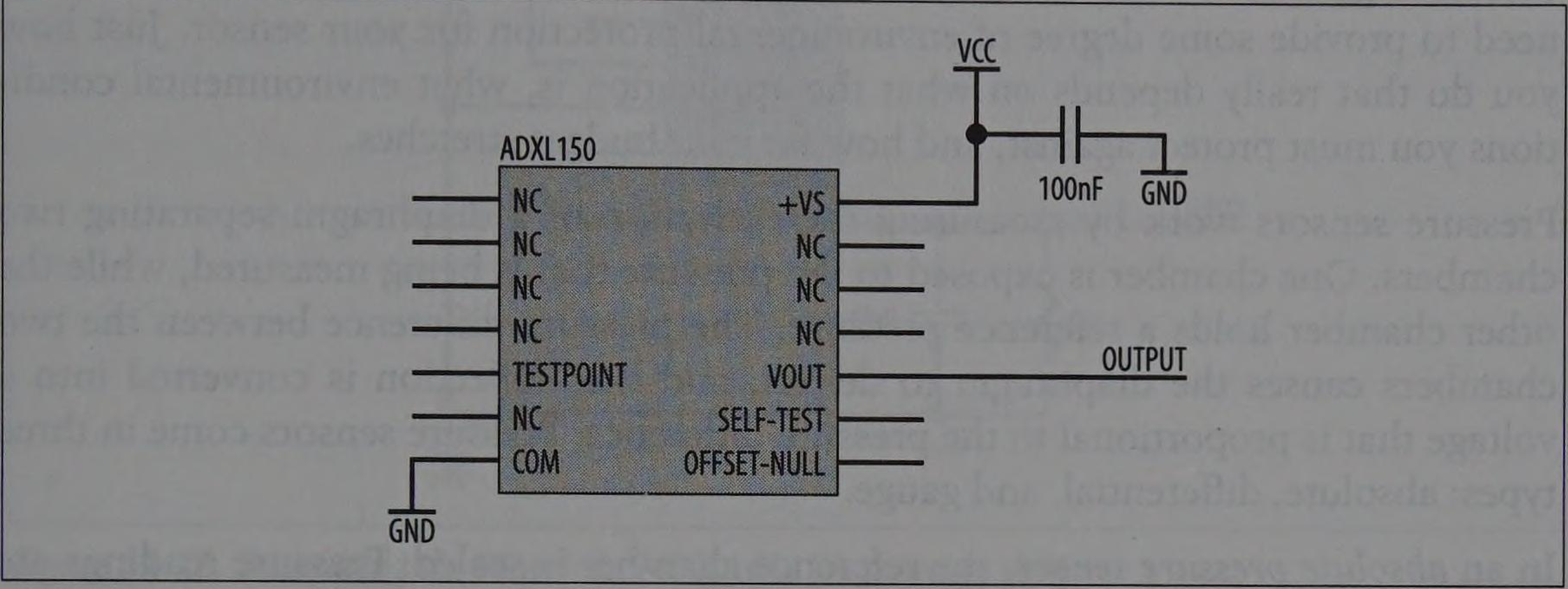


Figure 12-17. Using the ADXL150

Most of the pins are No Connection (NC) and can be ignored, as can the **TESTPOINT** and **SELF-TEST** pins. The **TESTPOINT** pin is used during manufacture only and should be left alone.

The ADXL150 operates off a power supply in the range of 4V to 6V. However, for ideal operation, the supply should be exactly 5.0V. The closer to 5V the supply is, the more accurate your measurements of acceleration will be. The output voltage is proportional to both acceleration and power supply (V_S) and is given by the relation:

```
VOUT = VS/2 - (sensitivity * VS/5 * acceleration)
```

The sensitivity value varies from device to device and is in the range 33.0 to 43.0, with a nominal value of 38.0. The standard sensitivity value gives a range of ±50g, however, the sensitivity may be doubled (giving a range of ±25g) by connecting the output to the **OFFSET-NULL** pin.

The **SELF-TEST** pin is used for verifying the correct operation of both the internal mechanics of the sensor as well as its signal conditioning and amplification electronics. Applying a logic 1 to this input pin artificially imposes a force on the sensor, and thus the sensor can be shown to be operating correctly.

Pressure Sensors

We're going to take a look at pressure sensors. The most obvious use is in measuring air pressure for weather monitoring and prediction. But pressure sensors are also used in cars to measure manifold pressure, in washing machines to measure water levels, and in biomedical applications, such as measuring blood pressure. Another application of pressure sensors is to measure altitude, since air pressure changes with height above sea level. Ocean depth can similarly be measured.

When using pressure sensors, the substance you are measuring can adversely affect the device. Remember that these are sensitive electronic components, and fluids or corrosive gases can destroy them. So unless you're measuring clean, dry air, you'll need to provide some degree of environmental protection for your sensor. Just how you do that really depends on what the application is, what environmental conditions you must protect against, and how far your budget stretches.

Pressure sensors work by measuring the deflection of a diaphragm separating two chambers. One chamber is exposed to the pressure that is being measured, while the other chamber holds a reference pressure. The pressure difference between the two chambers causes the diaphragm to deflect, and this deflection is converted into a voltage that is proportional to the pressure difference. Pressure sensors come in three types: absolute, differential, and gauge.

In an *absolute pressure sensor*, the reference chamber is sealed. Pressure readings are referenced to an absolute pressure, hence the name. Absolute sensors normally have the reference chamber pressure at vacuum or at one atmosphere.

In a *differential sensor*, the reference chamber is not sealed, and an external pressure reference may be applied. Differential sensors are used to measure the relative pressures between two gases or two liquids. A differential sensor may be treated as an absolute sensor by providing it with a sealed and stable reference pressure.

A *gauge sensor* is a variation of the differential pressure sensor, where the reference pressure chamber is open to the atmosphere. Thus, the measured pressure is referenced to atmospheric pressure, and variations of atmospheric pressure (such as those caused by weather conditions or altitude) are taken into account. One interesting use of a gauge pressure sensor is to measure airspeed. If the measuring chamber is exposed to the oncoming airflow (caused by the aircraft's motion), and the reference chamber is exposed to the air but sheltered from the effects of the airflow, then the difference in pressure can be used to calculate the airspeed of the aircraft.

So, with all that in mind, let's take a look at some pressure sensors. The first sensor is a Motorola MPXA6115A absolute pressure sensor (Figure 12-18). It operates from a 5V supply and will give an output voltage of between 0.2V and 4.8V, proportional to pressures of 15kPa to 115kPa. (*Pa* is *Pascals* and is a unit of pressure.) Unlike most other pressure sensors, which require external signal conditioning, temperature compensation, and signal amplification, the MPXA6115A integrates it all in one neat little package. It comes in an eight-pin chip package, with or without snorkel!

The NC pins are no connection and should be left unwired. The only additional components required are a decoupling capacitor on the power supply and a resistor and capacitor in parallel at the output. The output may be directly connected to an ADC's input.

The second pressure sensor we will look at is also an absolute pressure sensor. But, unusually, rather than producing an analog output, it incorporates a built-in ADC. It

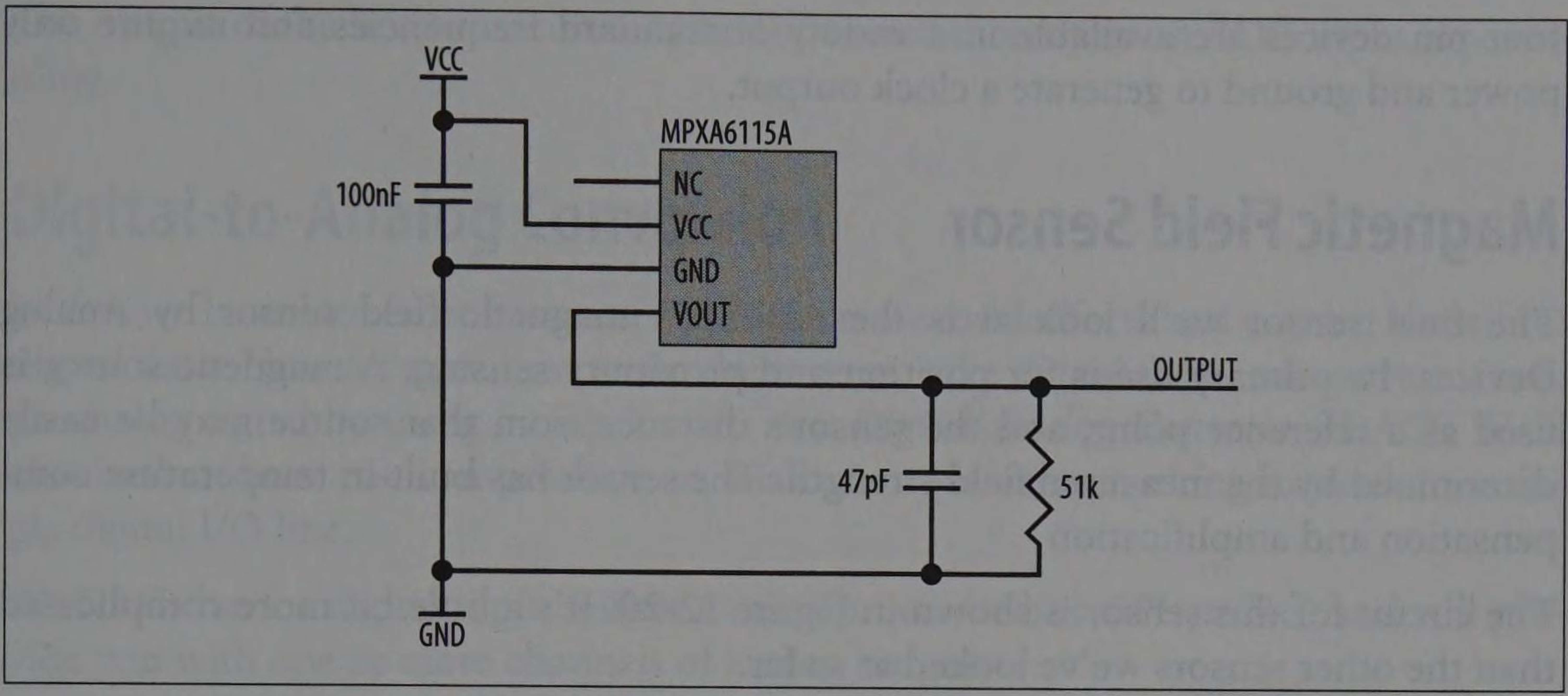


Figure 12-18. Interfacing the Motorola MPXA6115A pressure sensor

is interfaced to a microcontroller using SPI, and being digital, it is much less susceptible to noise and interference. The sensor is the KP100, made by Infineon Technologies (*http://www.infineon.com*) in Munich, Germany.

The schematic for a circuit based upon the KP100 is shown in Figure 12-19.

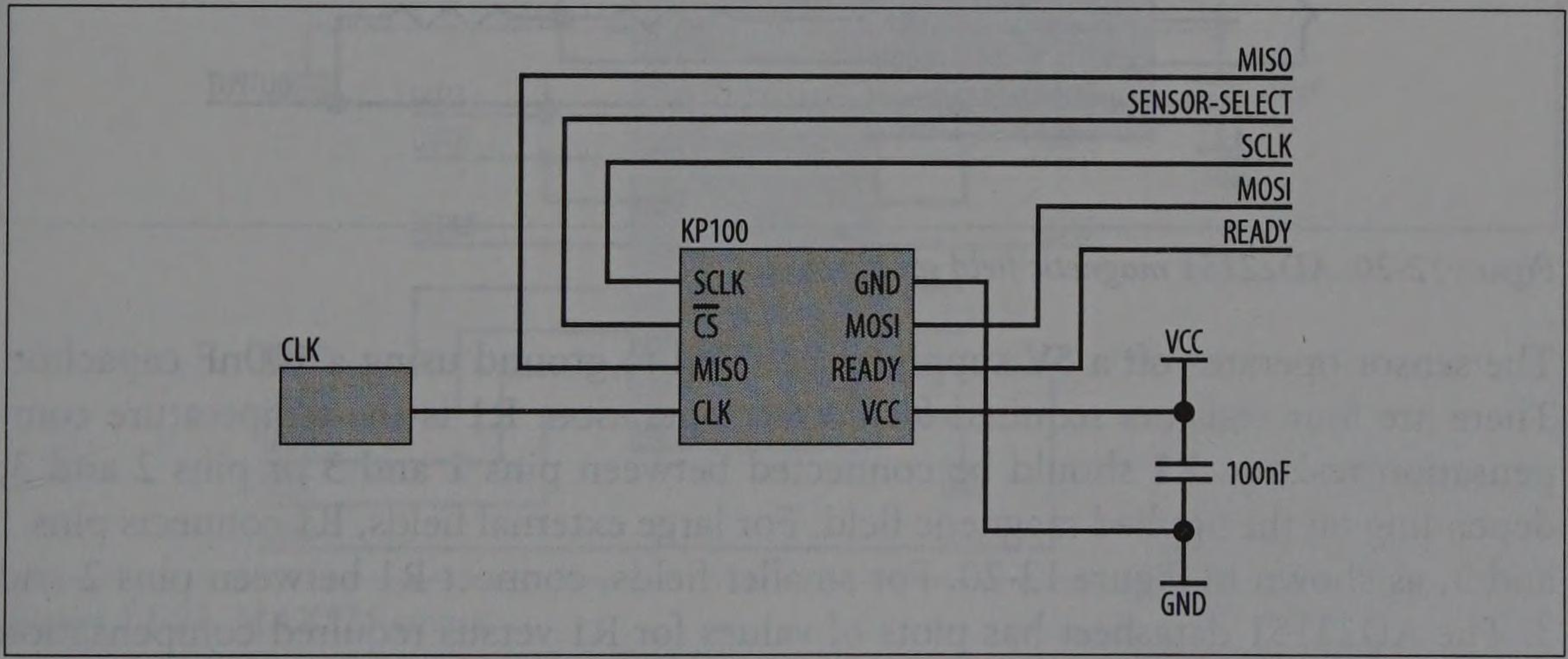


Figure 12-19. KP100 pressure sensor circuit

The sensor operates off a 5V supply, and this is decoupled to ground using a 100nF capacitor to reduce noise. The sensor has a standard SPI-style interface and is connected to a microcontroller as with any SPI device. The sensor also provides a **READY** output, which may be used to interrupt the host processor or may simply be connected to a spare I/O and read as a digital status flag. The KP100 also requires a separate clock (**CLK**) input. This clock can be either 4MHz or 8MHz. If the processor is running at one of these speeds, then the sensor can share the same clock input as the processor. However, if the processor is operating at a different clock frequency, the KP100's clock may be easily generated using a clock module. These

four-pin devices are available in a variety of standard frequencies and require only power and ground to generate a clock output.

Magnetic Field Sensor

The final sensor we'll look at is the AD22151 magnetic field sensor by Analog Devices. Its primary use is for position and proximity sensing. A magnetic source is used as a reference point, and the sensor's distance from that source may be easily determined by the measured field strength. The sensor has built-in temperature compensation and amplification.

The circuit for this sensor is shown in Figure 12-20. It's a little bit more complicated than the other sensors we've looked at so far.

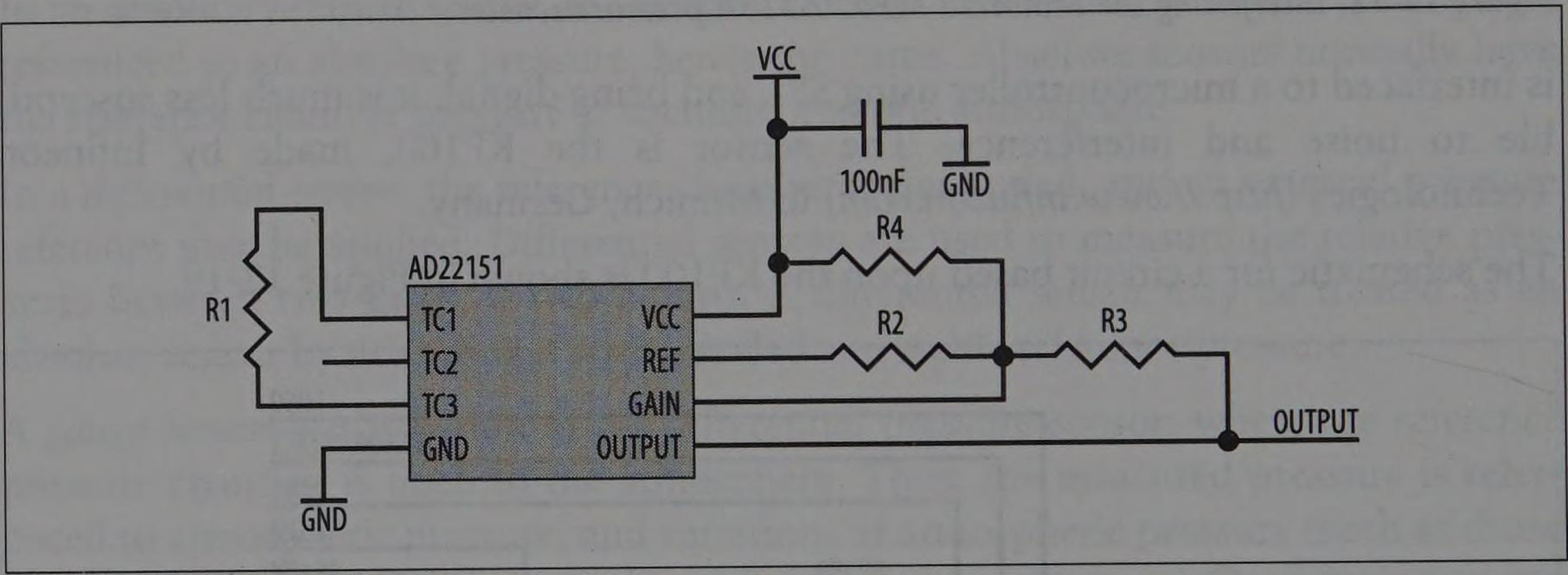


Figure 12-20. AD22151 magnetic field sensor circuit

The sensor operates off a 5V supply, decoupled to ground using a 100nF capacitor. There are four resistors required for correct operation. R1 is the temperature compensation resistor. R1 should be connected between pins 1 and 3 or pins 2 and 3, depending on the applied magnetic field. For large external fields, R1 connects pins 1 and 3, as shown in Figure 12-20. For smaller fields, connect R1 between pins 2 and 3. The AD22151 datasheet has plots of values for R1 versus required compensation levels. Check with the manufacturer of your magnetic source* as to the required compensation value, and use this in conjunction with the datasheet to determine R1.

For magnet data, try *http://www.magtech.com.hk*, *http://www.eastindustries.net*, or *http://www.millennium.com.hk* as places to start.

R2 and R3 set the signal gain of the internal amplifier, and R4 provides a voltage offset. The datasheet for the sensor contains equations and technical data for computing values of these resistors, based upon your specific needs.

* I'm assuming here that you are using a magnet specifically intended for such applications and which has data available, rather than a magnet you've found lying around somewhere.

The output of the sensor circuit may be connected directly to an ADC input for sampling.

Digital-to-Analog Conversion

So far, we have looked at how you can sense real-world effects and convert these into digital data. Now let's see how to do the reverse: take digital data and convert it into an analog signal by using a chip known as a *Digital-Analog Converter* (*DAC*). We'll also look at how you can produce an analog output using nothing more than a single digital I/O line.

All DACs have a digital input (either a microprocessor bus, SPI, or I2C) and will provide you with one or more channels of analog output.

The Maxim MAX525 is a 12-bit DAC that interfaces to a host processor using SPI. It has four channels of analog output and incorporates output amplifiers on-chip. The inverting inputs of both amplifiers are accessible so that you can alter their respective gains. An example circuit for a MAX525 is shown in Figure 12-21.

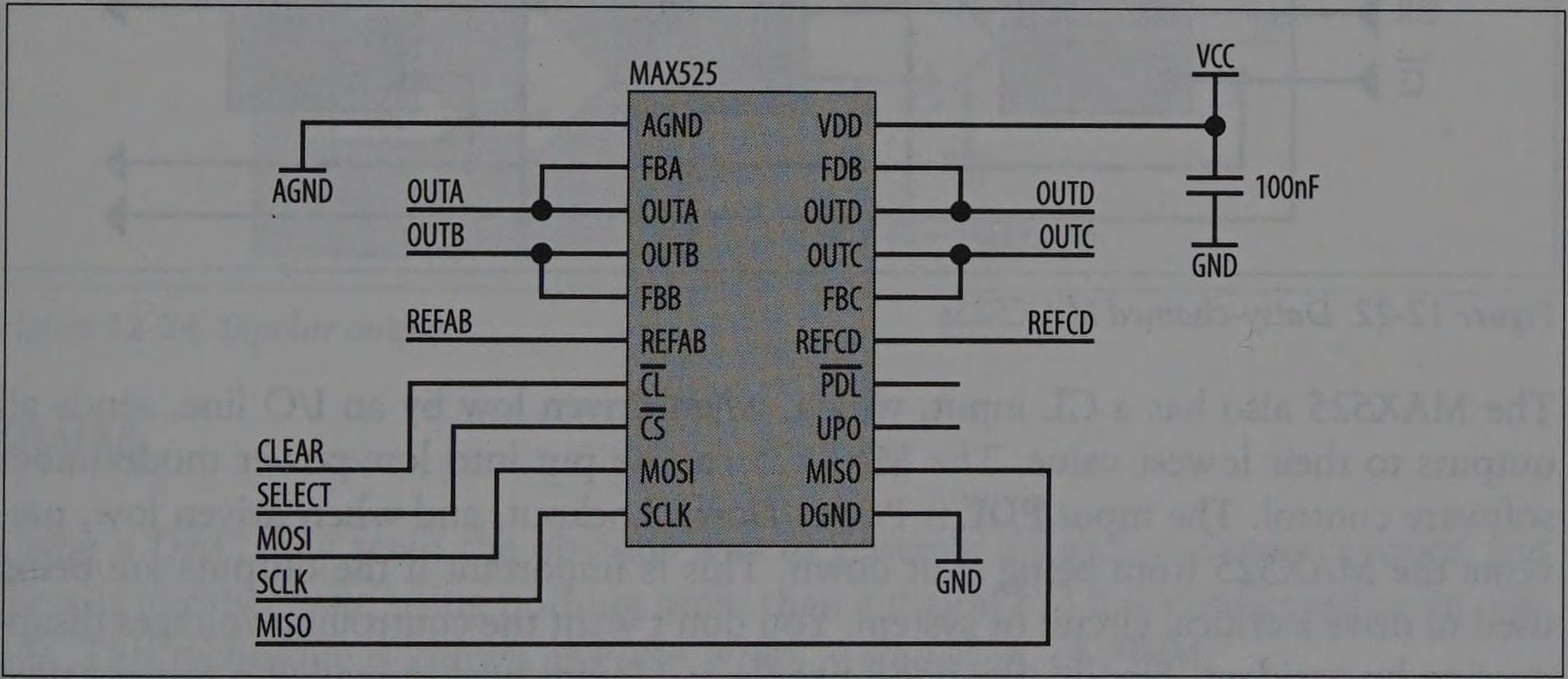


Figure 12-21. MAX525 circuit

The four analog output channels are **OUTA**, **OUTB**, **OUTC**, and **OUTD**. These are tied directly to their respective feedback inputs (**FBA**, **FBB**, **FBC**, and **FBD**) for standard unipolar operation. There are two voltage reference inputs, **REFAB** (for channels A and B) and **REFCD** (for channels C and D). These two reference inputs must be at least 1.4V or more below **VCC** at all times. The output voltage for each channel is given by the relation:

V_{OUT} = (V_{REF} * code / 4096) * gain

where `code` is the digital value written to that channel. In our example circuit, the gain is 1. (See the earlier section on noninverting amplifiers for how to set gains.) If

our reference voltage is set to 3.6V, the digital value 4095 (0xFFF) generates an output voltage of:

```
VOUT = (VREF * 4095 / 4096) * gain
     = 3.6 * 0.9997 * 1
     = 3.59V
```

Similarly, the digital value 2048 (0x800) generates an output voltage of:

```
VOUT = (VREF * 2048 / 4096) * gain
     = 3.6 * 0.5 * 1
     = 1.8V
```

Note the separate analog and digital grounds in the schematic. These should be connected together, but only at a single point close to the DAC.

The MAX525 has a standard SPI connection to a microprocessor. Multiple MAX525s may be daisy-chained together for efficiency (Figure 12-22).

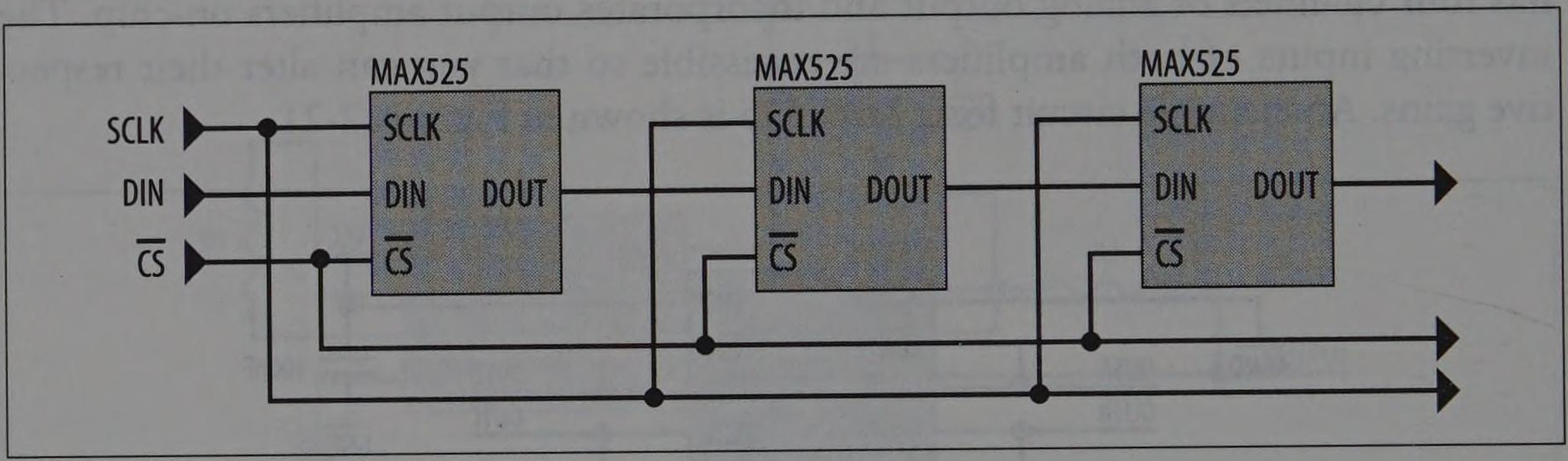


Figure 12-22. Daisy-chained MAX525s

The MAX525 also has a $\overline{CL}$ input, which, when driven low by an I/O line, sends all outputs to their lowest value. The MAX525 can be put into low-power mode under software control. The input **PDL** is Power-Down Lockout, and when driven low, prevents the MAX525 from being shut down. This is important if the outputs are being used to drive a critical circuit or system. You don't want the controlling voltages disappearing by accident. Finally, the good people at Maxim have provided a signal called **UPO** (User Programmable Output). This is a general-purpose output that can be driven high or low under software control. Use it for whatever purpose you require.

Now, if you want a gain other than 1 (nonunity gain), external resistors are required for the output amplifier. The schematic for this (for a single output channel) is shown in Figure 12-23.

From before, we have that the gain of a noninverting amplifier is given by:

```
Gain = 1 + R2 / R1
```

For bipolar output on a given channel, an external amplifier (with bipolar supplies) does the job (Figure 12-24).

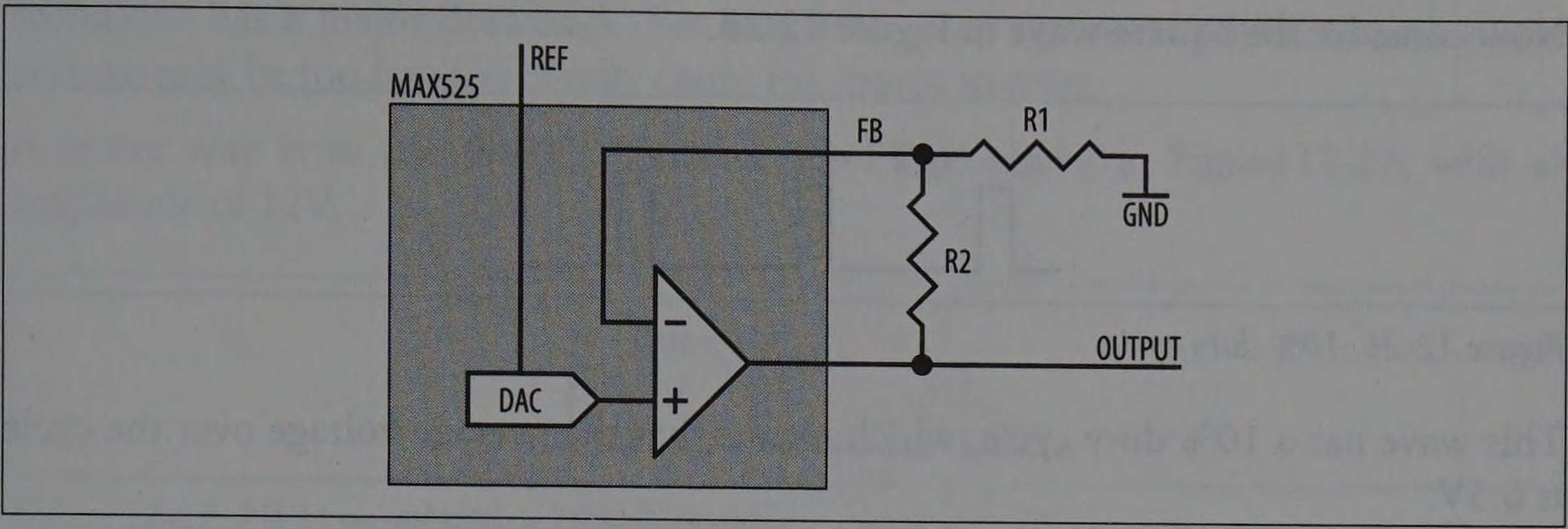


Figure 12-23. Feedback resistors for nonunity gain

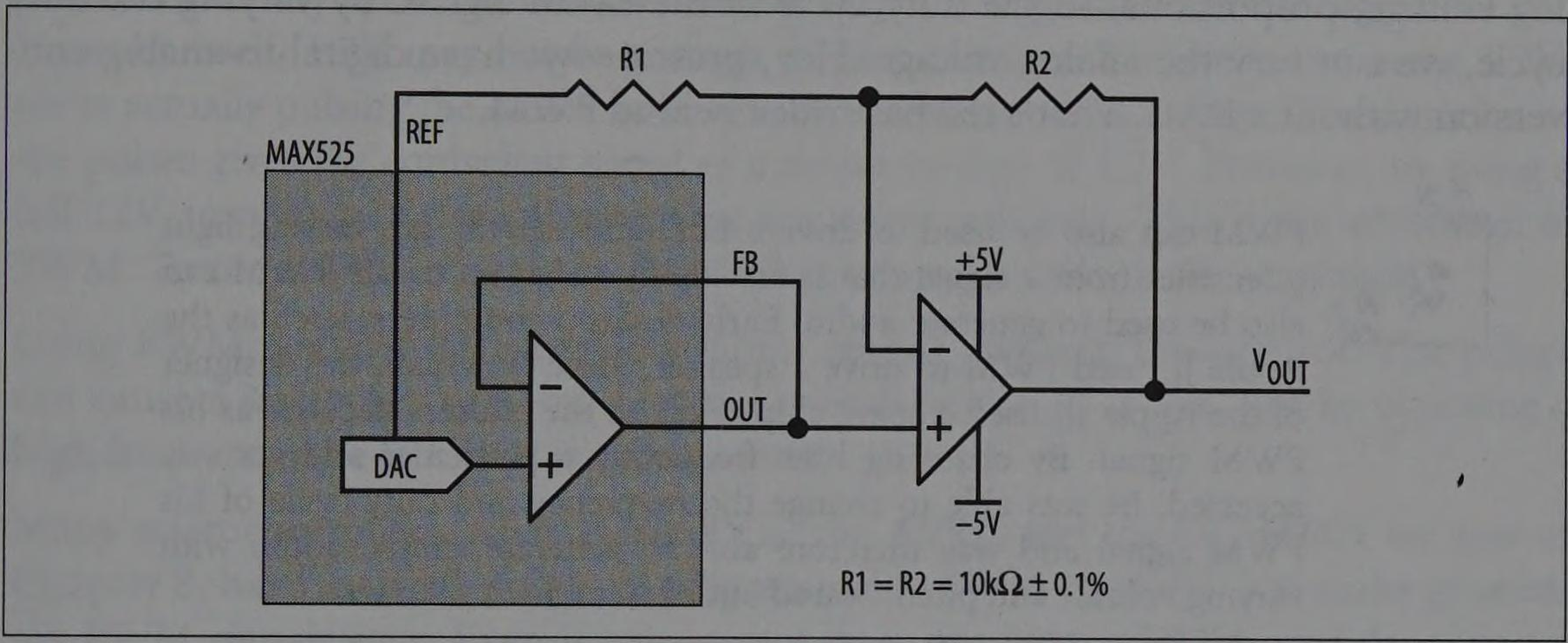


Figure 12-24. Bipolar output

PWM

Using a DAC may seem the obvious way to generate an analog output voltage. But there is another way, using nothing more than a digital I/O line configured as an output. This technique is known as *Pulse Width Modulation*, or *PWM*.

Consider the average, garden-variety square wave shown in Figure 12-25.

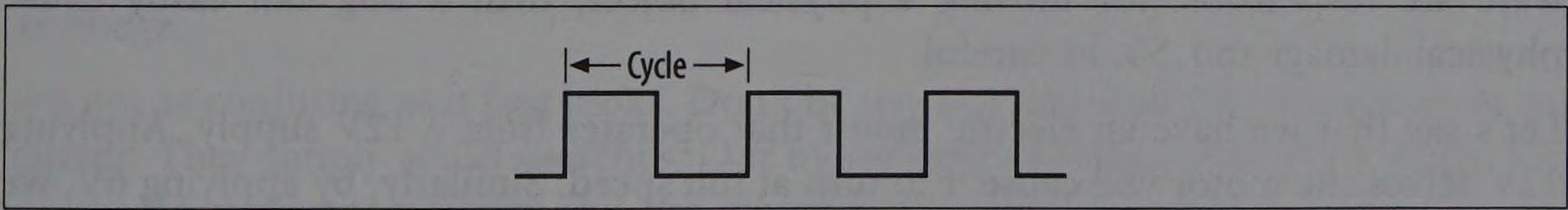


Figure 12-25. A ubiquitous square wave

The width of the high is equal to the width of the low, so this wave is said to have a 50% *duty cycle*. In other words, it is high for exactly half the cycle. Now, if the amplitude of this square wave is 5V, for example, the average voltage over the cycle is 2.5V. It is as though we had a constant voltage of 2.5V.

Now consider the square wave in Figure 12-26.

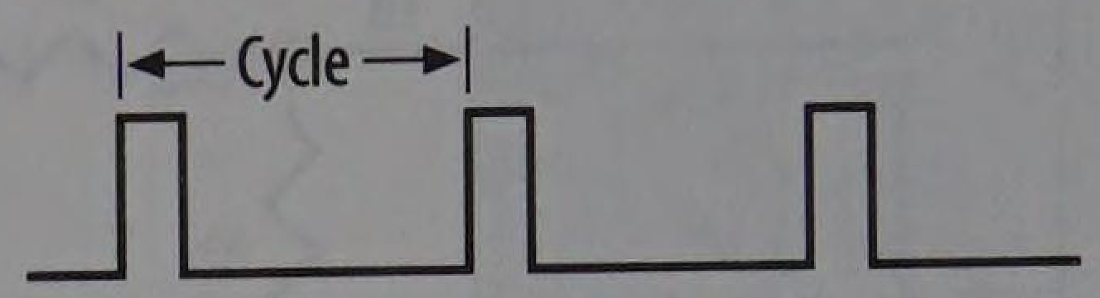


Figure 12-26. 10% duty cycle

This wave has a 10% duty cycle, which means that the average voltage over the cycle is 0.5V.

A low-pass (averaging) filter on the PWM output will convert the pulses to an analog voltage, proportional to the duty cycle of the PWM signal. By varying the duty cycle, we can vary the analog voltage. Hey, presto!—we have digital-to-analog conversion without a DAC. That's the basic idea behind PWM.

> PWM can also be used to drive a LED and thereby get varying light intensities from a signal that is essentially either on or off. PWM can also be used to generate audio. Early desktop computers, such as the Apple][, used PWM to drive a speaker. Steve Wozniak, the designer of the Apple][, used a spare chip select of the address decoder as his PWM signal. By changing how frequently a particular address was accessed, he was able to change the frequency and duty cycle of his PWM signal and was therefore able to generate simple audio with varying volume and pitch. Sound out of an address decoder!

Motor Control

One of the fun things you can do with an embedded computer is getting it to actually move something, whether it be an external system or the embedded computer itself. Motion implies motor, and this section will look at how you interface an embedded computer to an electric motor. The possible applications could range from controlling locomotives on your model railroad layout to experiments in robotics and anything in between. A note of caution though: if your hardware and software are responsible for moving a physical object, then a bug can easily cause physical damage too. So, be careful.

Let's say that we have an electric motor that operates from a 12V supply. Applying 12V across the motor will cause it to turn at full speed. Similarly, by applying 6V, we can get the motor spinning at half-speed. By varying the applied voltage, we can vary the speed at which the motor turns.

This voltage to drive the electric motor may be generated in several ways. The most obvious may seem to be to use a DAC to generate an analog output voltage and then use an amplifier to boost the signal to the voltage and current required to turn the motor. The speed of the motor is proportional to the output voltage. However, this

technique has a major drawback. For very low-speed operation, the required output voltage may be too low to actually cause the motor to turn.

A better way is to use PWM. Consider the PWM signal in Figure 12-27, with an amplitude of 12V.

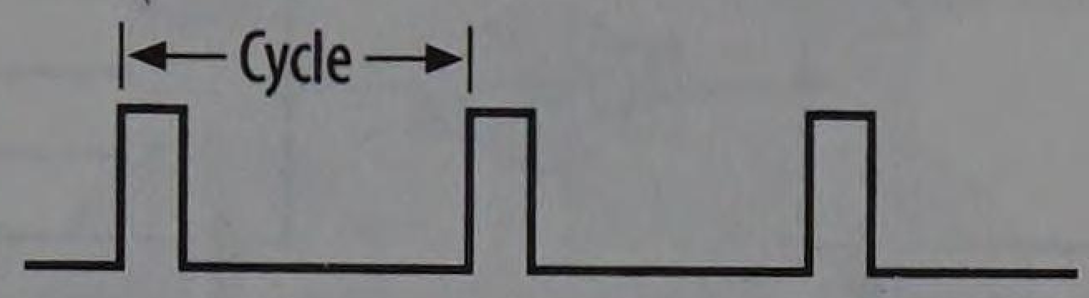


Figure 12-27. PWM signal with a 10% duty cycle

With a 10% duty cycle, the effective analog output voltage of this PWM signal is 1.2V. Now, by itself, 1.2V may not be enough to turn a motor. But, we're not using 1.2V; we're actually pulsing the motor with 12V, its maximum drive voltage. The duration of the pulses gives the equivalent speed of a motor voltage of 1.2V. However, by using a full 12V amplitude, we're ensuring that the motor will turn. This is the advantage of PWM. To control speed, we vary the width of the pulse and not the amplitude.

Using PWM, you can get very slow motor speeds and very fine control. The pulses can cause a jerkiness to the motor if the overall frequency is low, but by choosing a high frequency, the jerkiness is averaged out.

Many microcontrollers, such as the PICs, the AVRs, and the DSP56805 we saw in Chapter 8, have internal, software-programmable PWM modules that make generating PWM signals easy. Even if a processor does not have a PWM module, you can still generate PWM under software control, simply by using a digital output line.

Let's now take a look at how you would interface a processor to an electric motor using PWM. Due to the voltages and currents required by motors, you cannot simply hang a motor off the pins of a processor and expect it to work. You need an interface circuit that will take your logic-level, PWM output, and use this to switch much higher voltages and currents.

Figure 12-28 shows a conceptual model (in a crude and simplified form) of such an interface circuit, for driving a small electric motor. This type of circuit is known as an *H-bridge*.

It's not as confusing as it first looks. Don't be too worried about the transistors in the circuit. They simply act as switches. Our motor operates from a supply voltage, **V+**. Apply **V+** with one polarity and the motor turns in the forward direction. Reverse the polarity, and the motor reverses too. To drive the circuit, we use four outputs from the processor, two PWM (which I've called **PWM-A** and **PWM-B**), and two general I/O lines (which I've called **A** and **B**). Initially, all outputs are low, everything is turned off, and the motor is stationary.

If we send **A** high, the transistor Q4 turns on and connects the right "side" of the motor to ground. If we then send **PWM-A** high, the transistor Q1 turns on. Thus,

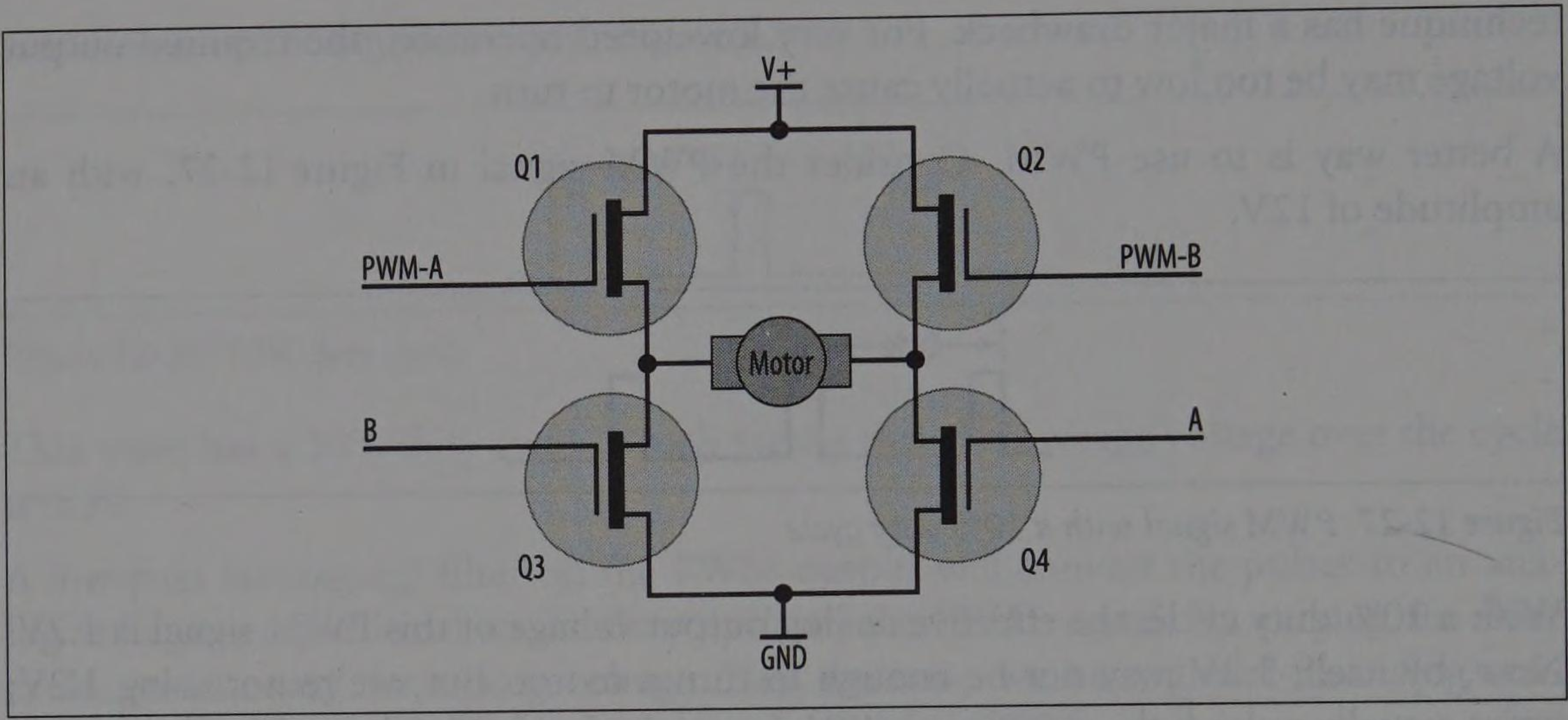


Figure 12-28. Motor drive circuit using an H-bridge

the left "side" of the motor is connected to **V+**, and the motor spins. By generating a PWM signal on **PWM-A**, we can control the speed of the motor in that direction.

Conversely, by leaving **A** and **PWM-A** low and setting **B** and **PWM-B** high, transistor Q2 and transistor Q3 turn on, and the motor spins in the reverse direction. By generating a PWM signal on **PWM-B**, we can control the speed in the reverse direction.

Care must be taken in your software. If both Q1 and Q3 are turned on or both Q2 and Q4 are turned on, then you effectively connect **V+** to ground, with very little resistance in between! The results would be spectacular and short-lived! A proper H-bridge circuit normally contains protection to prevent such a state from occurring.

The actual implementation of an H-bridge is a little more complicated and requires additional components such as protection diodes and so forth. Now, while you could design such an H-bridge circuit using discrete components, there is an easier way. A number of manufacturers, such as Motorola, International Rectifier (*http://www.irf.com*), M. S. Kennedy Corp (*http://www.mskennedy.com*), and others, make H-bridges in easy-to-use integrated circuits.

> If you're ever cruising around a component manufacturer's web site looking for devices that will switch high currents at high voltages, and you can't find them, scoot over to their "automotive components" section. Such devices are sometimes hidden away in there.

Let's look at an example H-bridge, the Motorola MC33186. This chip is more sophisticated than the simple H-bridge I used to explain the concept. It provides more functionality yet is easier to control. This chip can operate from a supply voltage (**V+**) of between 5V and 28V and can switch continuous currents as high as 5A, yet it has logic inputs that are compatible with TTL levels. It has built-in short-circuit and overcurrent protection. Figure 12-29 shows an MC33186 circuit.

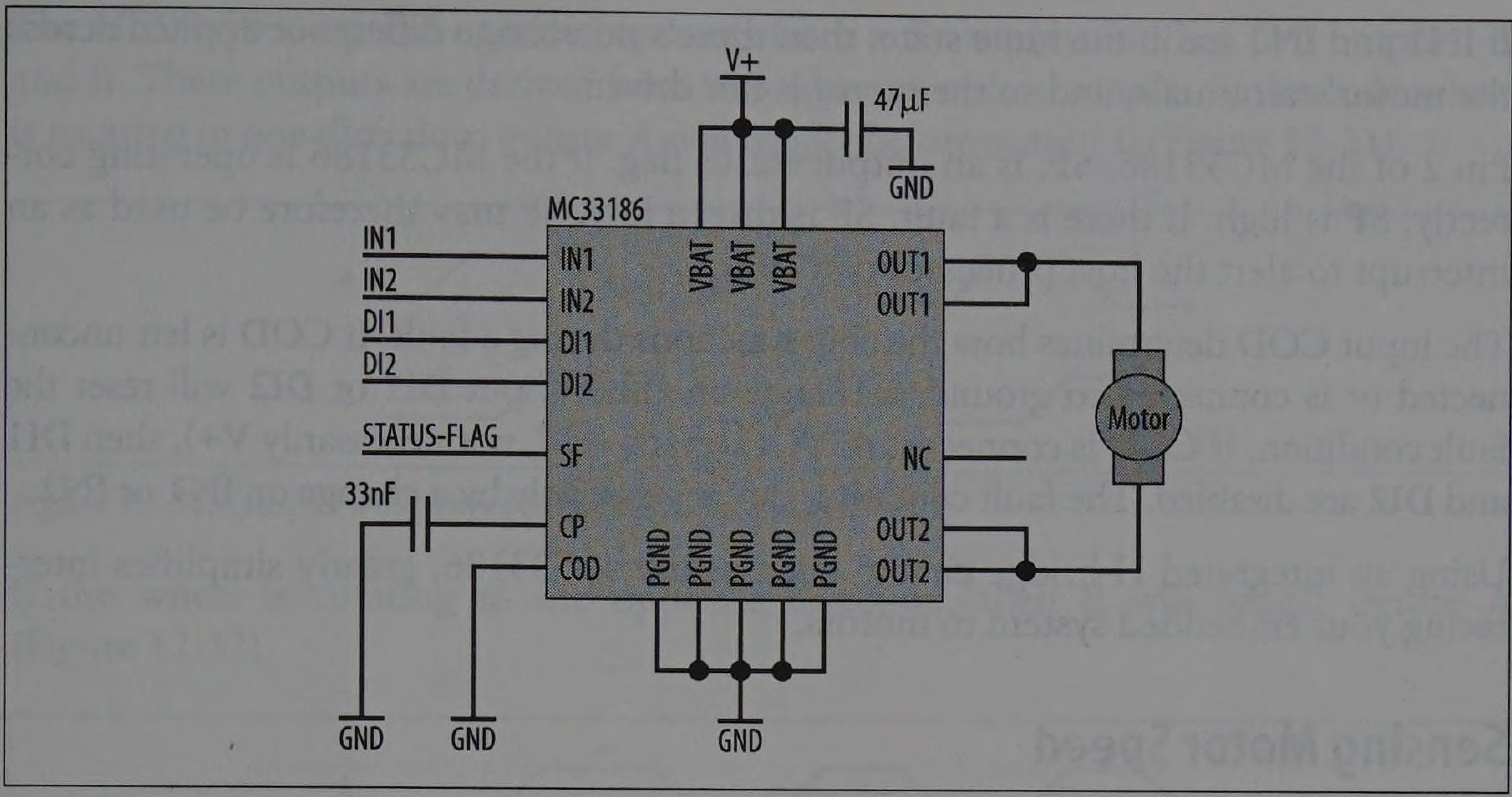


Figure 12-29. MC33186 motor drive circuit

The chip has three power-supply inputs, $\mathbf{V_{BAT}}$, all of which must be connected to the supply voltage, **V+**. The power-supply input needs to be decoupled using a 47µF capacitor. The internal charge pump also needs a decoupling capacitor. The pin, **CP**, provides access to the charge pump and requires a 33nF capacitor. The chip also has five ground pins, which, similarly, must all be connected to ground.

OUT1 and **OUT2** are the pins that directly drive the motor. There are two of each, so that the high output currents are not traveling through a single pin.

IN1 and **IN2** control both the motor's speed and direction. **DI1** and **DI2** serve to disable the MC33186. These four control signals may be driven by a microcontroller's I/O lines. For normal operation, **DI1** is low and **DI2** is high. Sending either **DI1** high or **DI2** low will disable the MC33186 and stop the motor. Table 12-1 shows how **IN1**, **IN2**, **DI1**, and **DI2** affect the motor's operation.

Table 12-1. MC33186 states of operation

DI1	DI2	IN1	IN2	OUT1	OUT2	Motor
Low	High	High	Low	V+	Ground	Forward
Low	High	Low	High	Ground	V+	Reverse
Low	High	Low	Low	Ground	Ground	Freewheeling
Low	High	High	High	V+	V+	Freewheeling
High	Don't care	Don't care	Don't care	High impedance	High impedance	Disabled
Don't care	Low	Don't care	Don't care	High impedance	High impedance	Disabled

If we want the motor to run forward, we generate a PWM signal on **IN1** and leave **IN2** low. If we want to run the motor backward, we leave **IN1** low and place a PWM signal on **IN2**. The duty cycle of the PWM signal determines the motor's speed. Simple.

If **IN1** and **IN2** are in the same state, then there's no voltage difference applied across the motor's terminals, and so the motor is not driven.

Pin 2 of the MC33186, **SF**, is an output status flag. If the MC33186 is operating correctly, **SF** is high. If there is a fault, **SF** is driven low. **SF** may therefore be used as an interrupt to alert the host processor of a fault.

The input **COD** determines how the chip functions during a fault. If **COD** is left unconnected or is connected to ground, a change on either input **DI1** or **DI2** will reset the fault condition. If **COD** is connected to **VCC** (that's +5V, not necessarily **V+**), then **DI1** and **DI2** are disabled. The fault condition can be reset only by a change on **IN1** or **IN2**.

Using an integrated H-bridge circuit, such as the MC33186, greatly simplifies interfacing your embedded system to motors.

Sensing Motor Speed

The system that the motor is driving will affect the motor's speed. If the motor must move a heavy load, then its actual speed of rotation may be less than the speed intended. In such situations, it is useful to measure the actual speed so that the embedded control system can compensate.

The easiest way to measure a motor's rotational speed is to use an optical encoder module, such as the Agilent HEDS-9000 or a similar device. The encoder consists of a light source (LED) and an array of photodetectors, separated by a slotted disk known as a *code wheel* (Figure 12-30). The disk is mounted on the rotating motor shaft. Each time a slot passes between the LED and a detector, the detector receives a flash of light and generates an electrical pulse. The rate at which the pulses are generated corresponds directly to the rotational speed of the motor. The resolution of the code wheel is known as its *counts per revolution*, or *CPR* value. The HEDS series of encoders are available with CPRs ranging from 96 all the way up to 2048.

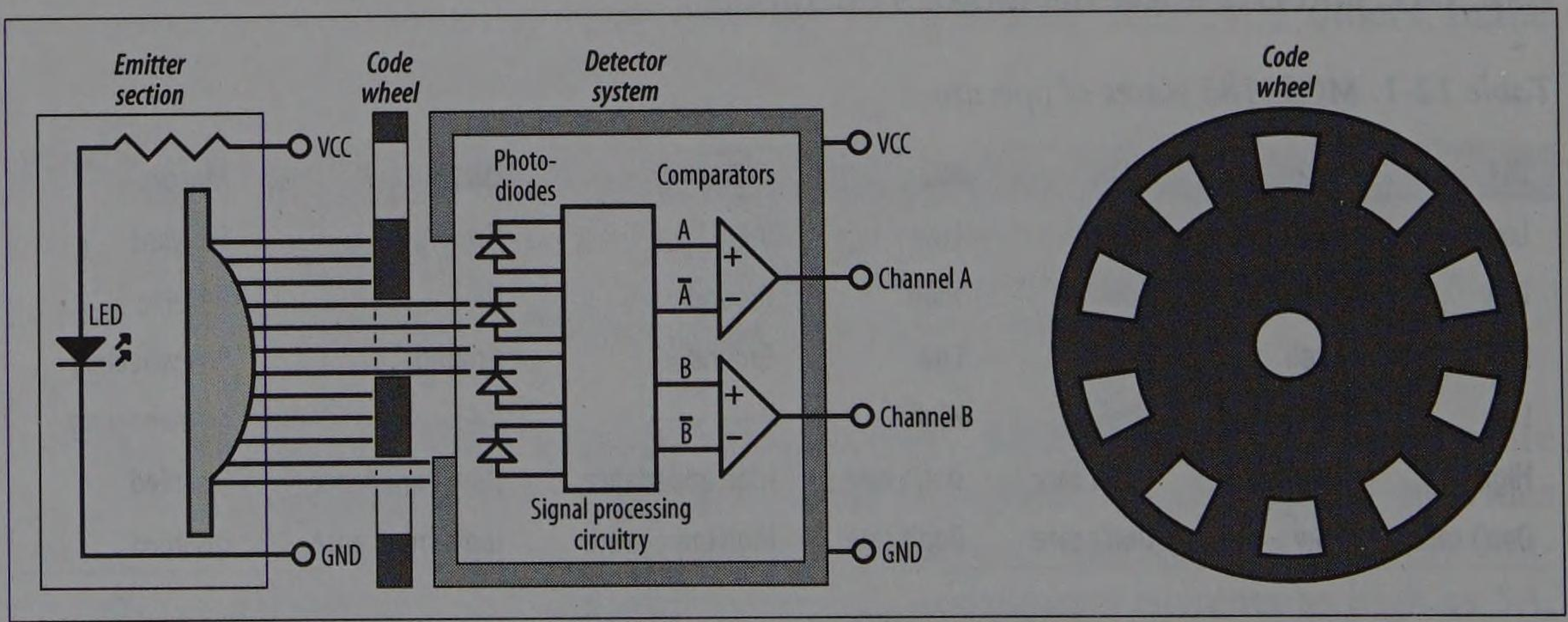


Figure 12-30. Block diagram of a HEDS-9000 optical encoder and a code wheel

The HEDS-9000 optical encoder operates from a 5V supply and has two outputs, **A** and **B**. These outputs are derived from two adjacent optical sensors. If the code wheel is rotating in one direction, output **A** will trigger before output **B** (Figure 12-31).

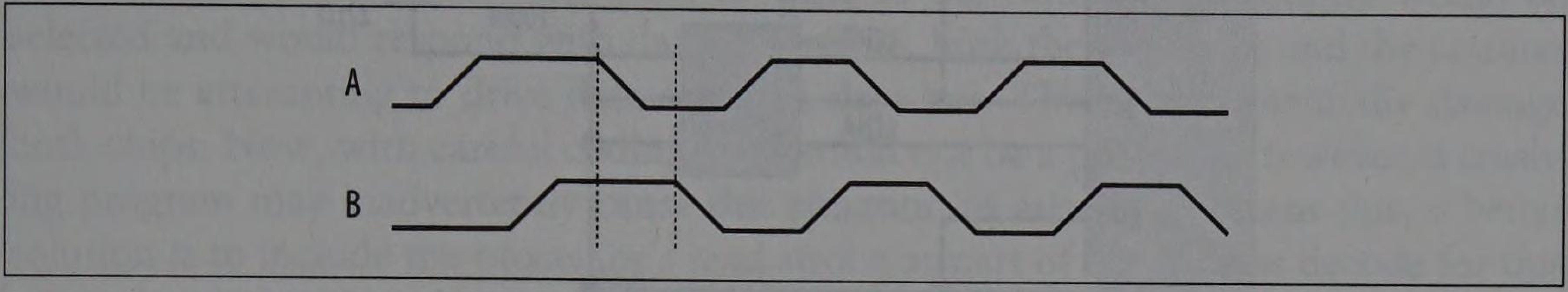


Figure 12-31. Output waveforms for the optical encoder

If the wheel is rotating in the opposite direction, then **B** will trigger before **A** (Figure 12-32).

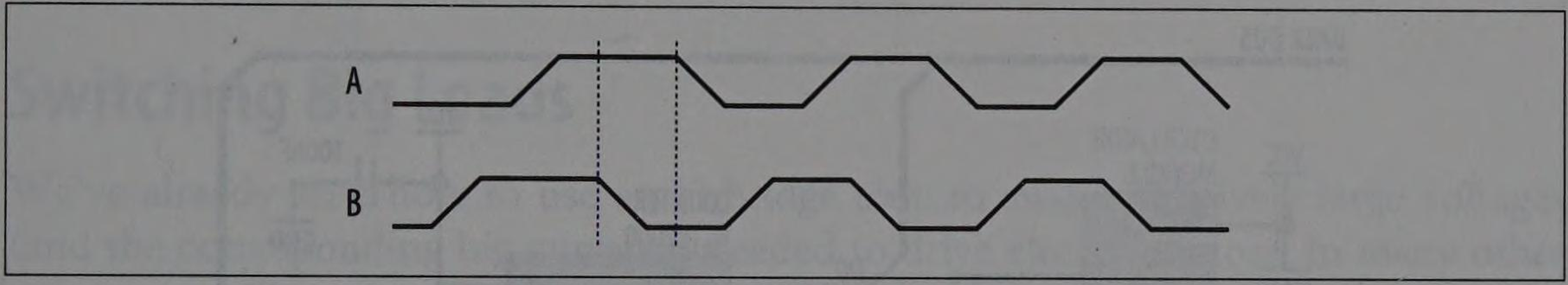


Figure 12-32. Output waveforms for the optical encoder

The rate at which the pulses arrive gives the motor's speed, and the order in which they arrive shows the direction. This is known as *quadrature encoding*.

Most microcontrollers have timer/counter inputs that can measure external trigger events such as these. Under software control, you can use the timers to monitor these quadrature signals. However, Agilent makes a series of devices known as *quadrature counters*, the 12-bit HCTL-2000, the 16-bit HCTL-2016, and the 16-bit, cascadable HCTL-2020. These chips provide a bus-based interface to a processor and convert quadrature signals into a binary number representing motor position. A 16-bit position counter is capable of measuring 32,767 increments in either direction, which corresponds to approximately 15 turns of a 2048 CPR encoder. To determine the present motor speed or position, the processor simply reads from the quadrature counter as though it were just another memory location. Quadrature counters also have noise filters on their inputs and so provide a more reliable, and more accurate, way of determining motor position.

The schematics showing an optical encoder and quadrature counter are shown in Figure 12-33 and Figure 12-34. The optical encoder is placed on a separate, small PCB so that it may be easily mounted next to the motor's shaft. The quadrature counter is located on the embedded computer's PCB. IDC headers (J1 and J2) and a ribbon cable connect the two circuit boards.

The quadrature counter requires a 14MHz clock. This is easily provided by an oscillator module. **CHA** and **CHB** are the quadrature inputs from the encoder. The counter has a reset input, $\overline{\textbf{RST}}$, which clears the counter. Asserting $\overline{\textbf{RST}}$ zeros the

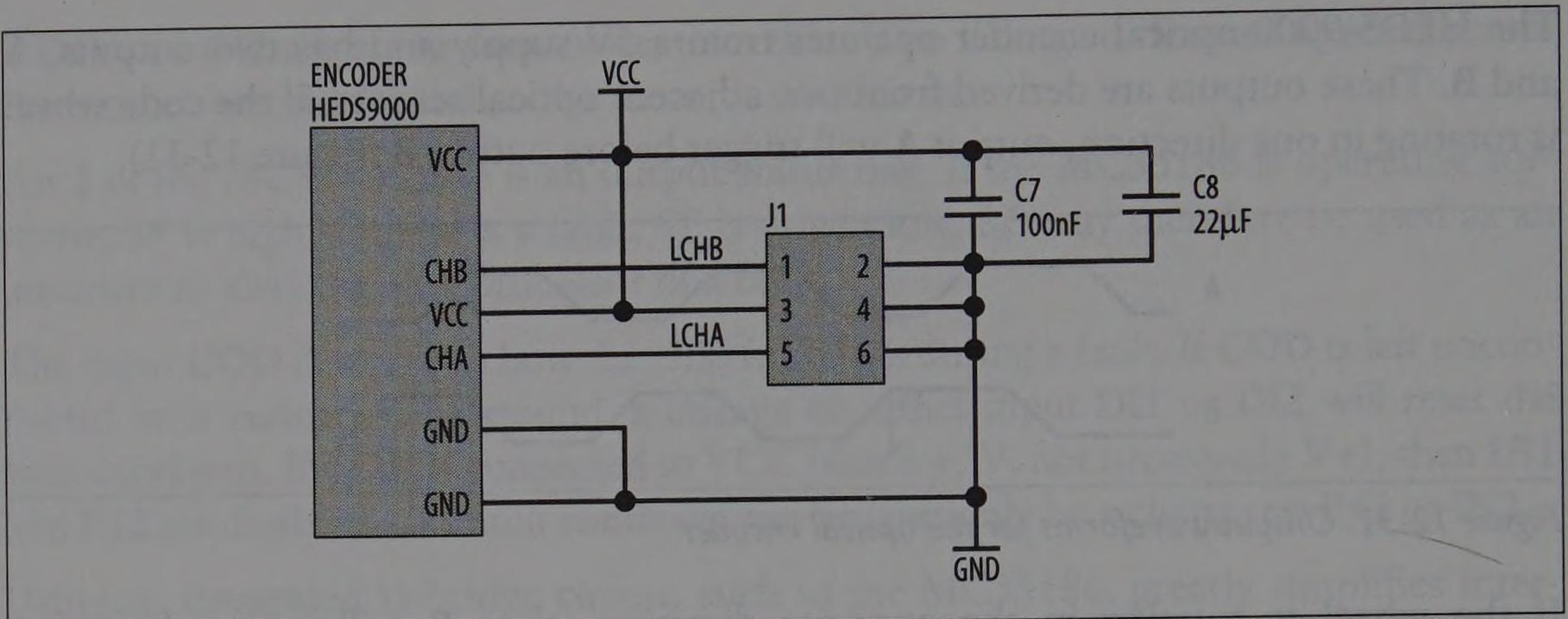


Figure 12-33. Optical encoder circuit

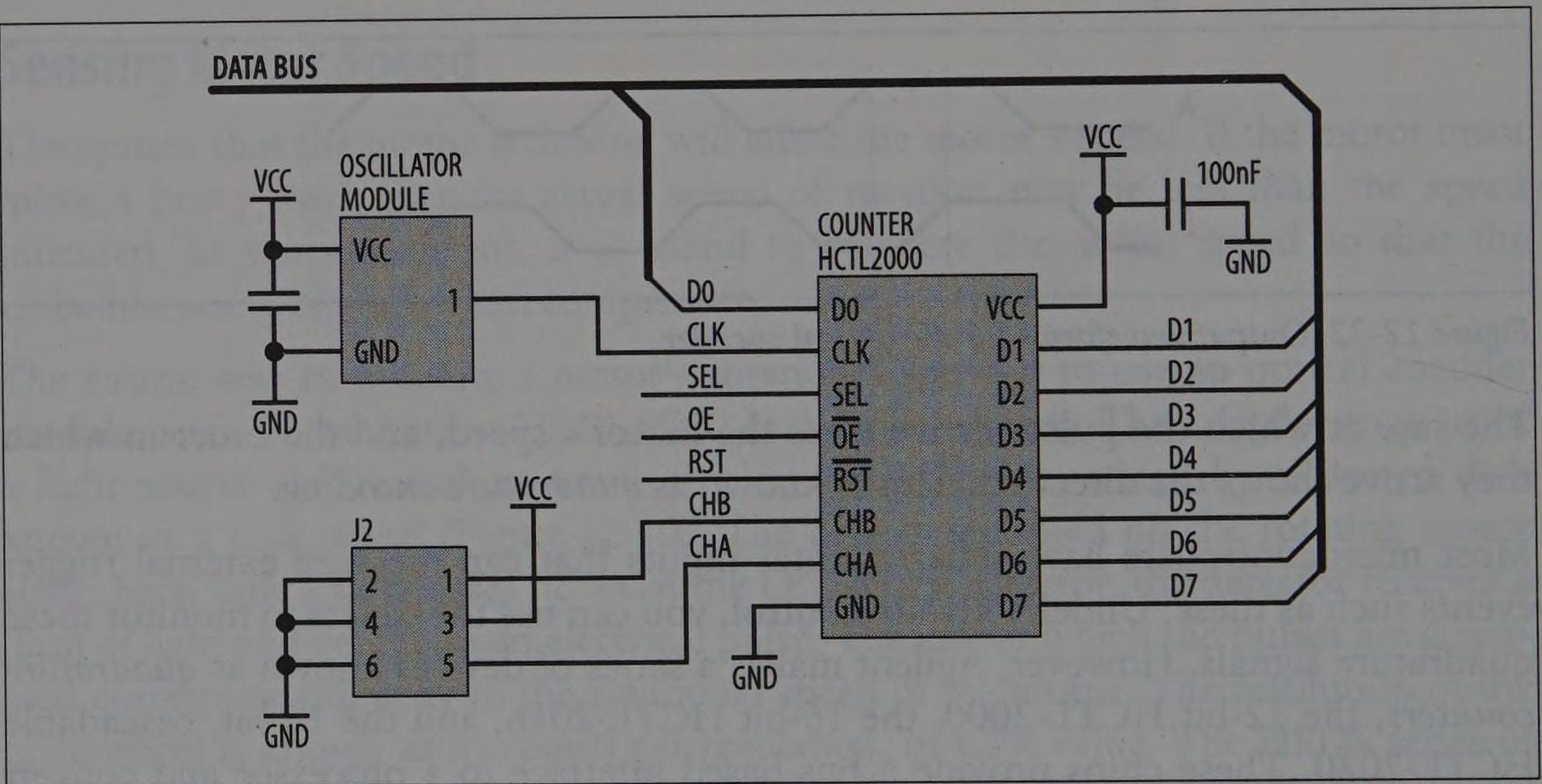


Figure 12-34. Quadrature counter circuit

quadrature counter and indicates that the motor is in the "home" position. This input is driven by a digital output of the microcontroller so that the counter can be reset under software control.

D0 to **D7** are the data bus through which the processor reads the current position. Since the counters are either 12 bits or 16 bits, two reads are necessary to retrieve the value through the 8-bit bus. The counter therefore occupies two locations in memory, and the **SEL** input is used to select which byte is being read. If **SEL** is low, then the higher-order bits are read. If **SEL** is high, then the lower-order bits are read. To make these 2 bytes appear in adjacent memory locations, the processor's address line, **A0**, is used to drive **SEL**. Thus, the least-significant address of the two selects the upper 8 bits, while the next address selects the lower 8 bits.

The counter does not have a chip select as such. Since it is a read-only device, the counter's output enable, $\overline{\textbf{OE}}$, functions as a combined chip select and output enable.

Therefore, this input is driven by the output of the address decoder that corresponds to the region of the address space to which the counter is mapped. When the processor reads from that address range, $\overline{\mathbf{OE}}$ is asserted, and the counter responds with data. Note that if the processor attempted to write to the counter, the counter would be selected and would respond with data. Therefore, both the processor and the counter would be attempting to drive data onto the data bus. This could potentially damage both chips. Now, with careful coding, this would not be a problem. However, a crashing program may inadvertently cause this situation to arise. To prevent this, a better solution is to include the processor's read strobe as part of the address decode for this particular device. In other words, the counter is selected only if both the address is correct *and* the processor is performing a read. If the processor is performing a write to the counter's address, the counter is not selected and the access is ignored.

A quadrature counter allows you to accurately monitor a motor's position and speed.

Switching Big Loads

We've already seen how to use an H-bridge chip to switch relatively large voltages (and the corresponding big currents) needed to drive electric motors. In many other cases, you will want to turn large voltages on or off, and in this section I will show you an easy way of doing just that.

The Motorola MC33298 is a chip that is controlled by a microprocessor using SPI and can switch eight power sources on or off. This chip can handle voltages between 5V and 26.5V, with currents as large as 6 Amps. If you need to turn electrical systems on or off, this chip is for you. Its primary use is for industrial and automotive applications, controlling power to subsystems such as heaters, small air-conditioning units, moderate voltage lightbulbs, small pumps, and so on. Obviously, it won't handle the high AC voltages that come out of your wall socket, so don't use it for switching power to your home appliances!

The basic schematic for the circuit is shown in Figure 12-35.

The MC33298 has two power-supply pins. **VDD** is a 5V supply and powers the chip's internal digital logic. It's decoupled to ground using a 100nF capacitor. $\mathbf{V_{PWR}}$ is the supply voltage for the external subsystems (represented in the figure by each LOAD rectangle) and can range from 5V to 26.5V. There are eight switch outputs, labeled **OUT0** through **OUT7**. When a given switch is activated, the corresponding output is connected through to the $\mathbf{V_{PWR}}$ supply, thereby turning that subsystem on. The MC33298 has short-circuit detection and shutdown (with automatic retry), overvoltage detection and shutdown, current limiting on the outputs, output clamping during inductive switching, and thermal shutdown if the device is dissipating too much power. Higher currents may be switched by tying two or more outputs together so that the current is shared by more than one pin. By tying all outputs together, currents as high as 48A may be switched, limited only by the total power dissipation and corresponding thermal shutdown limit.

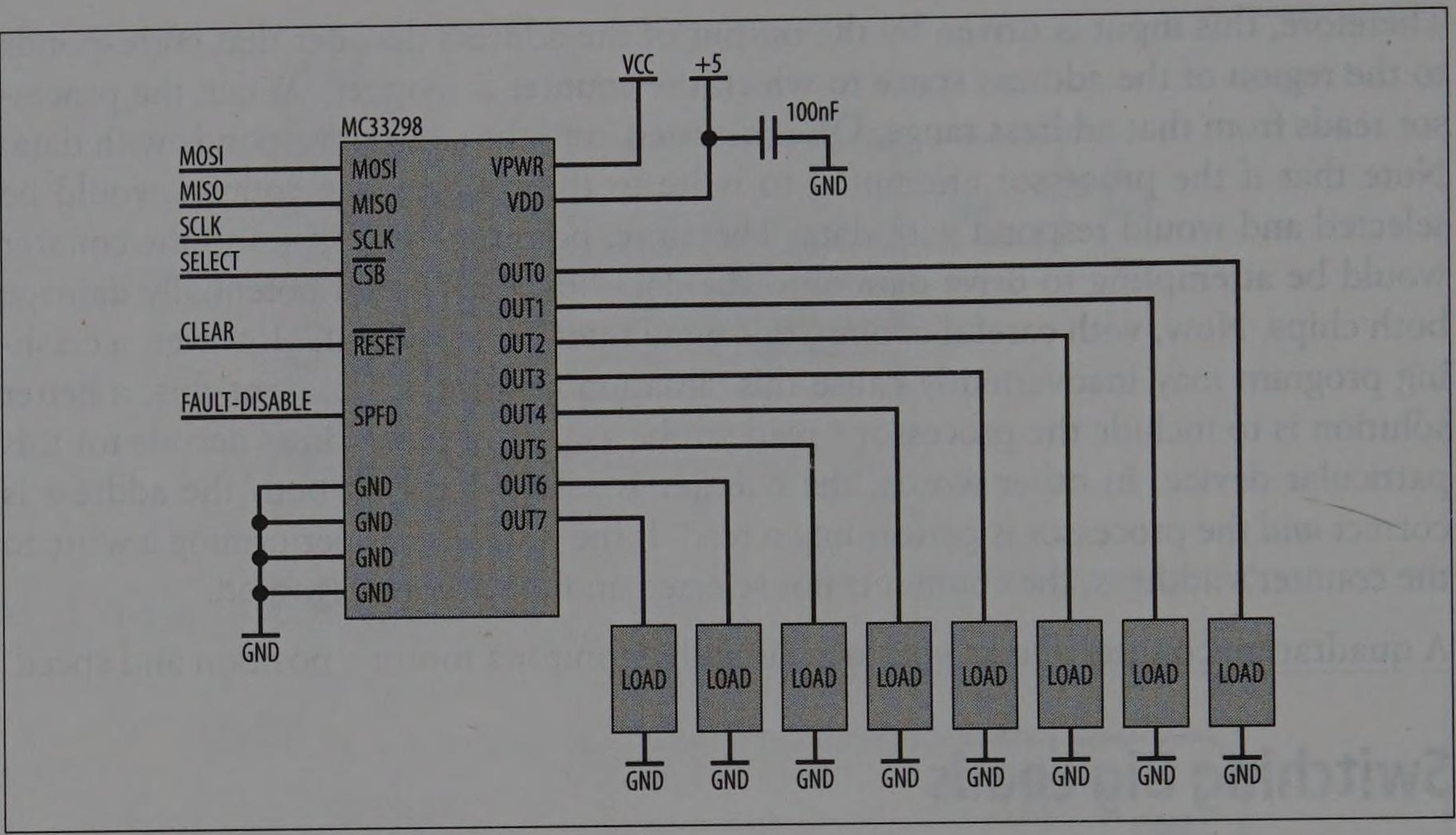


Figure 12-35. MC33298 circuit

The chip has a standard SPI port, allowing it to be interfaced to, and therefore controlled by, most microprocessors. The SPI signals **MOSI**, **MISO**, and **SCLK** are connected directly to a processor's SPI pins. The chip's select input, **$\overline{\text{CSB}}$**, is controlled by a digital output of the processor and is used to select the device during a SPI transfer. The device may be reset and all outputs turned off by asserting its **$\overline{\text{RESET}}$** input. Again, this too can be driven by a digital output of the processor so that the chip may be turned off under software control. The MC33298 supports SPI daisy-chaining, so multiple devices may be coupled together.

The **SPFD** pin is Short Fault Protect Disable. Sending this pin high allows the internal over-current detection circuitry to be disabled. When switching some loads, such as lightbulbs, there is a very high current for a short period of time. This would normally cause the MC33298 to register an overcurrent fault and shut that output off. The **SPFD** pin allows this protection to be overridden so that such loads may be controlled. Even though the overcurrent protection is bypassed, the MC33298 is still protected. If the high current lasts long enough, the chip's thermal shutdown circuit will kick in, thereby preventing damage. **SPFD** may be driven by a processor digital output and should be used with caution! For normal operation (with overcurrent protection on), this pin should be low.

And with that, we've reached the end of *Designing Embedded Hardware*. In this book, I have tried to introduce you to the basics of creating small computer systems, without getting bogged down in complicated detail. As a result, this book isn't the final word on computer electronics. It is merely the beginning, providing you with sufficient knowledge to read and explore further. Building you own computer hardware is rewarding and fun. I wish you the best of luck as you join the ranks of computer designers around the world.

References

Agere Systems Inc., *USS-820D USB Device Controller*, Allentown, PA, 2002

Agilent Technologies Inc., *Agilent HEDR-542x Series Optical Encoder Assemblies, Application Note 1267*, 2001

Agilent Technologies Inc., *Agilent HSDL-3310 IrDA Data Compliant 1.152 Mb/s Infrared Transceiver*, 2001

Agilent Technologies Inc., *Infrared IrDA Compliant Transceiver*, 1996

Agilent Technologies Inc., *Infrared Transceiver PC Board Layout for Noise Immunity*, 1999

Agilent Technologies Inc., *IrDA Physical Layer Implementation for Agilent Technologies' Infrared Products*, 1999

Agilent Technologies Inc., *Quadrature Decoder/Counter ICs Technical Data*

Agilent Technologies Inc., *Resolving IrDA Physical Layer Implementation Problems*, 1999

Agilent Technologies Inc., *Two Channel Optical Incremental Encoder Modules Technical Data*

Analog Devices Inc., *ADXL150/ADXL250 Datasheet*, Norwood, MA, 1998

Analog Devices Inc., *AD22103*, Norwood, MA, 1995

Analog Devices Inc., *AD22151*, Norwood, MA, 1997

Analog Devices Inc., *AD5203*, Norwood, MA, 1998

Analog Devices Inc., *AD620*, Norwood, MA, 1999

Analog Devices Inc., *AD623*, Norwood, MA, 1999

Atmel Inc., *AT25080 Series*, 2002

Atmel Corp., *AT25F512*, San Jose, CA, 2001

Atmel Corp., *AT43312A*, San Jose, CA, 2001

Atmel Corp., *AT45DB041*, San Jose, CA, 2001

Atmel Inc., *AT45DB161*, 2001

Atmel Corp., *AT76C711*, San Jose, CA, 2001

Atmel Inc., *AT90S515*, 2001

Atmel Corp., *AT90S8515 Preliminary*, San Jose, CA, 1998

Atmel Corp., *ATtiny15L*, San Jose, CA, 2001

Atmel Corp., *AVR040: EMC Design Considerations*, San Jose, CA, 2000

Atmel Corp., *AVR109: Self-programming*, San Jose, CA, 2001

Atmel Corp., *AVR180: External Brown-Out Detection*, San Jose, CA, 1998

Atmel Corp., *AVR300: Software I²C Master Interface*, San Jose, CA, 2002

Atmel Corp., *AVR460: Embedded Web Server*, San Jose, CA, 2002

Atmel Corp., *AVR910: In-System Programming*, San Jose, CA, 2002

Au, E., and Y. Zeng, *AVR*, Atmel Corp., Hong Kong, 1999

Axelson, J., *Designing RS-485 Circuits*, Circuit Cellar, June 1999

Axelson, J., *USB Chip Choices*, Circuit Cellar, July 2000

B&B Electronics, *RS-422 and RS-485 Application Note*, Ottawa, IL, 1992

Bagschik, P., *An Introduction to CAN*, *http://www.ime-actia.com/can_intro.html*, 2001

Ball, S., *Build an AVR Programmer*, Circuit Cellar, February 2000

Bogen, A., and V. Wollan, *AVR Enhanced RISC Microcontrollers*, Atmel Corp., Trondheim, Norway

Catsoulis, J., *Microprocessors*, Embedded Pty. Ltd., Brisbane, Australia, 1998

Compaq, HP, Intel et al., *Universal Serial Bus Specification*, 2000

Crystal Semiconductor Corp., *Crystal LAN CS8900 Ethernet Controller Technical Reference Manual*, Cirrus Logic, Austin, TX, 1996

Crystal Semiconductor Corp., *Using the Crystal CS8900 in 8-Bit Mode*, Cirrus Logic, Austin, TX, 2000

Cypress Semiconductor Corp., *CY62256*, San Jose, CA, 2002

Dallas Semiconductor Corp., *DS1307*, 1995

Fairchild Semiconductor Corp., *MC78xx/LM78xx/MC78xxA Datasheet*, 2001

Gergen, J., P. Hoang, and E. Chemaly, *Novel Digital Signal Processing Architecture with Microcontroller Features*, Motorola Inc., Austin, TX

Goldie, J., *Ten Ways to Bulletproof RS-485 Interfaces*, National Semiconductor, 1998

Harrison, R., *The Ins and Outs of RS232*, Australian Electronics Monthly, November 1985

Hitachi Ltd., *HM62V8512A Series*, Tokyo, 1996

Hitachi Semiconductor, *HM628511H Series*, 1998

Horowitz, P. and W. Hill, *Art of Electronics*, Cambridge University Press, 1989

Intel Corp., *High Speed USB Platform Design Guidelines*, 2000

Infineon Technologies AG, *KP100*, Munich, 1999

Knutson, C., and G. Diviney, *Infrared Data Communications with IrDA*, Counterpoint Systems Foundry Inc.

Maxim Inc., *Choosing the Right Power-Supply ICs*, Sunnyvale, CA, 2001

Maxim Inc., *MAX1245*, Sunnyvale, CA, 1996

Maxim Inc., *MAX1615*, Sunnyvale, CA, 1998

Maxim Inc., *MAX3100*, Sunnyvale, CA, 2001

Maxim Inc., *MAX3110E/MAX3111E Datasheet*, Sunnyvale, CA, 1999

Maxim Inc., *MAX3222/MAX3232/MAX3237/MAX3241*, Sunnyvale, CA, 1996

Maxim Inc., *MAX525*, Sunnyvale, CA, 1996

Maxim Inc., *MAX603/MAX604*, Sunnyvale, CA, 1994

Maxim Inc., *MAX6662*, Sunnyvale, CA, 2001

Maxim Inc., *MAX724*, Sunnyvale, CA, 1995

Maxim Inc., *Proper Wiring for RS-485*, Sunnyvale, CA, 2001

Megowan, P., D. Suvak, and C. Knutson, *IrDA Infrared Communications—an Overview*, Counterpoint Systems Foundry Inc.

Microchip Technology Inc., *24LC164*, Chandler, AZ, 1995

Microchip Technology Inc., *Low Power Design Using PIC16/17*, Chandler, AZ, 1995

Microchip Technology Inc., *PIC16/17 Oscillator Design Guide*, Chandler, AZ, 1994

Microchip Technology Inc., *PIC16C7X Data Sheet*, Chandler, AZ, 1998

Millar, I., et al., *The IrDA Standards for High-Speed Infrared Communications*, The Hewlett-Packard Journal, February 1998

Motorola Inc., *DSP56F805*, Austin, TX, 2002

Motorola Inc., *Interfacing Serial LCDs to a DSP56F805's SCI and Porting Code Using the Embedded SDK*, Denver, 2001

Motorola Inc., *MC33186*, Austin, TX, 2000

Motorola Inc., *MPXA6115A/D*, Denver, 2001

Motorola Inc., *Programming On-Chip Flash Memories of DSP56F80x DSPs Using JTAG/OnCE Interface*, Denver, 2002

Philips Semiconductors, *PCF8583*, Philips Electronics N. V., Eindhoven, The Netherlands, 1997

Samsung Electronics Co. Ltd., *T6T0808C1D Family*, 1997

Stuart, R., *IrDA Control*, Sharp Microelectronics,1999

TAOS Inc., *TSL250R, TSL251R, TSL252R*, TAOS Inc., Plano, TX, 2001

Texas Instruments Inc., *SN74HCT139*, Dallas, 1997

Texas Instruments Inc., *SN74HCT573*, Dallas, 1998

Texas Instruments Inc., *SN74HCT574*, Dallas, 1997

Williams, S., and I. Millar, The IrDA Platform, HP Laboratories, Bristol

Williamson, T., *Designing Microcontroller Systems for Electrically Noisy Environments*, Intel Inc., 1982

Zalac, D., *DSP56800 Hardware Interfacing Techniques*, Motorola Inc., Austin, TX, 2001

Zerkus, M., J. Lusher, and J. Ward, *USB Primer*, Circuit Cellar, May 1999

Index

A

absolute pressure sensor, 268
AC (Alternating Current), 39, 40
accelerometer, 266
accumulators, 14
adapters, power, 68
ADC (Analog-to-Digital Converter), 255–259
  interfaces, 259–261
adding
  displays, 206
  interfaces
    Ethernet, 246–251
    LAN, 238
  memory to SPI, 197–199
  processors, 100
  RTC, 205
  status LEDs, 121–122
address
  bus, 12
  decoders, 133, 138–144, 162–164
  decoding, 12
  general call, 204
  logical, 149
  physical, 149
  space, 11
  translation, 154
addressing modes, 15
aliasing, 257
ALU (Arithmetic Logic Unit), 13–14
Amperes, 37
amplifiers, 252–255, 265
analog signals, 38–39
  amplifiers, 252–255
  converting, 255–259, 271–273
  switching, 122–125
Analog-to-Digital Converter
  (see ADC)
analysis, timing, 146–148
anode, 52
applications
  interrupts, 18
  layers, 5
architecture, 3–4
  AVR, 116–118
  buses, 11
  DMA, 26–32
  DSP, 21, 170–172
  DSP56800, 173–175
  embedded computer, 32–35
  Harvard, 11
  I/O, 26
  load/store, 21
  memory, 22–26
    EEROM, 25
    EPROM, 25
    Flash, 25
    RAM, 23–24
    ROM, 24
  Motorola 68000, 157–161
  parallel, 29
  PIC, 108
  processors
    addressing modes, 15
    ALU, 13–14
    big-endian/little-endian, 16–17
    functions, 13

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

architecture, processors (*continued*)
registers, 14
stacks, 15
Von Neumann machine, 10–11
Arithmetic Logic Unit (ALU), 13–14
assemblers, 7
assembly language, 6–8
asynchronous processors, 133
asynchronous transmission (UARTs), 210
ATMEL AT90S4434 processor, 128
ATMEL AT90S8515 processor, 116–118
bus signals, 134–138
memory cycle, 132–134
ATMEL AT90S8535 processor, 128
Attachment Unit Interface (AUI), 246
ATtiny15 processor, 118–125
analog signals, 122–125
status LEDs, 121–122
AUI (Attachment Unit Interface), 246
avoiding noise, 77–82
AVR
architecture, 116–118
ATtiny15 processors, 118–125
adding status LEDs, 121–122
switching analog signals, 122–125
processors, 127–128
ATMEL (see entries under ATMEL)
downloading code, 125–127

B

banked memory, 149, 151
batteries, 69–70
big-endian schemes, 16–17
binary numbers, xvii
bipolar signal, 38
blocking capacitors, 44, 48
BOD (brownout detector), 119
bootloader, 5
boundary scan, 175, 183
breadboarding, 82–83
bridge rectifier, 53, 69
brownout detector (BOD), 119
brownouts, 71
building
circuits, 82–83
adding processors, 100
debugging, 100–103
laying out PCBs, 87–90
obtaining PCBs, 83–87
routing, 91–94
testing, 96–99
PCBs, 91–94

bulk transfer, 229
burning the ROM, 24
bus enumeration, 229
bus interfacing, 11, 128–155
memory cycles, 132–134
memory management, 149–155
memory maps, 138–144
PALs, 144–146
signals, 134–138
timing analysis, 146–148
timing diagrams, 128–132
(see also USB)
bus-powered devices, 231

C

caches, 24
calculating power usage, 39
calendars, 190–194
CAN (Controller Area Network), implementing, 241–245
capacitive coupling, 79
capacitors, 43–45
decoupling, 74, 80
RC circuits, 45–49
cathode, 52
CCR (condition-code register), 14
Central Processing Unit (CPU), 9
ceramic capacitors, 44
charge pumps, 70, 71
chip select, 63
chips, xiv
circuits, 37
amplifiers, 252–255
boundary scan, 183
building, 82–83
laying out PCBs, 87–90
obtaining PCBs, 83–87
debugging, 100–103
diodes, 52–55
I2C, 201–205
displays, 206
RTC, 205
processors, 100
RC, 45–49
reset
DSP56805-based computers, 178–179
Motorola 68000, 162
routing, 91–94
testing, 96–99
troubleshooting ESDs, 81
voltage, 281–282

CISC (Complex Instruction Set Computer), 19–21
Clear To Send (CTS), 214
clocks, 56–59
  RTC, 191, 205
  SPI, 190–194
CMRR (Common-Mode Rejection Ratio), 255
code, 6, 125–127
codec, 255
cold weld, 83
common-mode rejection, 218
Common-Mode Rejection Ratio (CMRR), 255
compilers, 7
components
  connections, 90
  crystals, 55–59
  datasheets, 66
  decoupling capacitors, 74
  footprints, 85
  inductors, 49–51
  layers, 84
  passive (see passive components)
  populating, 88
  resources, xvi–xvii
  schematics, 61–64
computers, 28–32
  architecture, 3–4
  concept of, 4–22
    architecture, 8–17
    CISC, 19–21
    DSP, 21
    interrupts, 17–19
    processors, 6–8
    RISC, 19–21
  DSP56805, 175–183
    interrupts, 178–179
    memory, 179–183
    oscillator, 176
    reset circuit, 178–179
  embedded architecture, 32–35
  MIMD, 30–32
  Motorola 68000, 156–157
    address decoder, 162–164
    architecture, 157–161
    design, 161–169
    I/O, 164–165
    memory, 165–166
    reset circuit, 162
    wait states, 167–169
  SIMD, 29
concept of computers, 21
condition-code register (CCR), 14
conductors, 37
configuring PAL, 145
connections
  boundary scan, 183
  components, 90
  I2C, 201–205
    displays, 206
    RTC, 205
  IR, 220–226
    devices, 225–226
    IrDA, 220–224
    IrDA interfaces, 224–225
  Motorola 68000, 165–166
  RS-232C, 211–218
    handshaking, 214–215
    implementing, 215–217
    power supply, 217–218
  RS-422, 218–220
  series, 41
  USB, 226–237
    implementing, 234–237
    interfaces, 231–234
    packets, 229–231
  (see also routing)
constant voltage, 40
construction
  circuits, 82–83
    laying out PCBs, 87–90
    obtaining PCBs, 83–87
  debugging, 100–103
  PCBs, 91–94, 96–99
  processors, 100
  routing, 91–94
  testing, 96–99
consumption, reducing power, 58
control bus, 12
control, motor, 274–281
control transfer, 229
Controller Area Network (CAN), implementing, 241–245
controllers
  DSPs, 170–172
    DSP56800, 173–175
    DSP56805, 175–183
      JTAG, 183
  Program Controller, 173
  (see also AVR; PIC)
conversion
  analog-to-digital, 255–259
  DC voltage, 68–69

conversion (*continued*)
digital to analog, 271–273
opcodes, 7
regulators, 70–76
CPU (Central Processing Unit), 9
CRC (Cyclic Redundancy Check), 211
cross talk, 79
crystals, 55–59
CTS (Clear To Send), 214
current, 36–38, 59
diodes, 52–55
minimizing loop areas, 79
quiescent, 71
cycles, memory, 132–134
Cyclic Redundancy Check (CRC), 211

D

DAC (Digital-Analog Converter), 271–273
daisy-chaining SPI devices, 189
data bus, 12
Data Communication Equipment (DCE), 212
Data Terminal Equipment (DTE), 212
Data Terminal Ready (DTR), 217
data-chaining transfers, 27
datasheets, 66
bus interfacing, 128–155
memory cycles, 132–134
memory management, 149–155
memory maps, 138–144
signals, 134–138
timing analysis, 146–148
timing diagrams, 128–132
DC (Direct Current), 39, 40
DCE (Data Communication Equipment), 212
debouncing, 124
debugging, 95–96, 100–103
JTAG, 175, 183
decoders
addresses, 12, 133, 138–144
Motorola 68000, 162–164
decoupling
AC and DC components, 44
capacitors, 74, 80
demand-mode transfer, 27
descriptor, 231
design, xiv
avoiding noise, 77–82
circuits, 82–83
adding processors, 100
debugging, 100–103
laying out PCBs, 87–90
obtaining PCBs, 83–87
routing, 91–94
testing, 96–99
datasheets, 66
debugging, 95–96
DSP56805-based computers, 175–183
interrupts, 178–179
memory, 179–183
oscillator, 176
reset circuit, 178–179
Motorola 68000, 161–169
power
batteries, 69–70
converting DC voltage, 68–69
regulators, 70–76
rule check, 86
(see also AVR; PIC)
desktop computers
architecture, 3–4
concept of, 4–22
architecture, 8–17
CISC, 19–21
DSP, 21
interrupts, 17–19
processors, 6–8
RISC, 19–21
(see also computers)
development, IDE, 8
devices, IR, 225–226
differential pair, 218
differential sensor, 268
digital potentiometer, 194–197
Digital Signal Processors (see DSPs)
digital signals, 59–61
avoiding noise, 77–82
converting, 255–259, 271–273
logic analyzers, 95–96
(see also signals)
Digital-Analog Converter (DAC), 271–273
diodes, 52–55
directives (see schematics)
distortion, 253
distributed computers, 28–32
distribution of power, 79–81
DMA (Direct Memory Access), 26–32
DMAC (DMA controller), 27
documentation of datasheets, 66
downloading code, AVR processors, 125–127
downstream connections (USB), 232
DRAM (dynamic RAM), 22

DSPs (Digital Signal Processors), 170–172
  architecture of, 21, 170–172
  DSP56800, 173–175
  DSP56805, 175–183
    interrupts, 178
    memory, 179–183
    oscillator, 176
    RESET, 178
  JTAG, 183
DTE (Data Terminal Equipment), 212
DTR (Data Terminal Ready), 217
dynamic RAM (DRAM), 22
dynamic resistance, 55

E

EEROM (Electrically Erasable Read-Only Memory), 25
electric motor control, 274–281
Electrically Erasable Read-Only Memory (EEROM), 25
electrolytics, 45
electromagnetic effects, troubleshooting, 77–82
electromagnetic interference (EMI), 78
electronics
  capacitors, 43–45, 45–49
  crystals, 55–59
  current, 36–38
  diodes, 52–55
  inductors, 49–51
  resistors, 39–42, 45–49
  schematics, 61–64
  transformers, 51–52
  voltage, 36–38
    analog signals, 38–39
    power, 39
electrostatic discharge (ESD), 78, 81
embedded computers
  architecture, 3–4, 32–35
  concept of, 4–22
    architecture, 8–17
    CISC, 19–21
    DSP, 21
    interrupts, 17–19
    processors, 6–8
    RISC, 19–21
EMI (electromagnetic interference), 78
EPROM (Erasable Programmable Read-Only Memory), 25
error detection, 210
ESD (electrostatic discharge), 78, 81
Ethernet, implementing, 245–251
events, 17–19
exceptions, 17
external ADCs, 259–261
external memory
  AVR bus interfacing, 128–155
  DSP56805-based computers, 179–183
  Motorola 68000, 165–167

F

Farads, 43
fast hardware interrupt, 18
Field-Programmable Gate Arrays (FPGAs), 118
File Select Register (FSR), 108
firmware, 5, 24
Flash, 25, 256
floating, 59
flow control, 214–215
fly-by transfer, 27
footprints, 85
forward voltage drop, 52, 121
four-wire interface (see SPI)
FPGAs (Field-Programmable Gate Arrays), 118
frequency, 38, 253
FSR (File Select Register), 108
functions
  I/O, 164–165
  of processors, 13

G

gauge sensor, 268
general call address, 204
generation of interrupts, 17–19
ground, 37
  bounce, 81
  loops, 80
  plane, 79
grounding mats, 82

H

handshaking, 214–215, 231
hardware
  debouncing, 124
  interrupts, 17–18
  looping, 22
Harvard architecture, 11
Henries, 49
hexadecimal numbers, xvii

high impedance, 59
high-pass filter, 48
holes, 37
hot-carrier diodes, 55

I

I2C (Inter Integrated Circuit), 201
  displays, 206
  RTC, 205
IACK (Interrupt Acknowledge), 164
ICE (In-Circuit Emulator), 96
ICP (In-Circuit Programming), 24
IDE (Integrated Development Environment), 8
implementing
  CAN, 241–245
  Ethernet, 245–251
  IrDA, 220–224
  RS-232C interfaces, 215–217
  RS-422, 218–220
  RS-485, 238–241
  USB, 234–237
In-Circuit Emulator (ICE), 96
In-Circuit Programming (ICP), 24
index registers, 14
inductors, 49–51
  transformers, 51–52
infrared (IR), 220–226
  devices, 225–226
  IrDA, 220–224
  IrDA interfaces, 224–225
inline assembly, 8
input capacitance, 47
instructions
  assembly language, 7
  processors, 6
  Return from interrupt, 17
insulators, 37
In-System Programming (ISP), 24
Integrated Development Environment (IDE), 8
Inter Integrated Circuit (see I2C)
interfaces
  accelerometer, 266
  ADCs, 259–261
  AUI, 246
  bus, 128–155
    memory cycles, 132–134
    memory management, 149–155
    memory maps, 138–144
    PALs, 144–146
    signals, 134–138
    timing analysis, 146–148
    timing diagrams, 128–132
  Ethernet, 246–251
  IR, 220–226
    devices, 225–226
    IrDA, 220–224, 224–225
  LAN, 238
  motor control, 274–281
  RS-232C, 211–218
    handshaking, 214–215
    implementing, 215–217
    power supply, 217–218
  RS-422, 218–220
  SPI, 125, 187–201
    clock/calendar, 190–194
    memory, 197–199, 200–201
    potentiometer, 194–197
  SSI, 175
  UARTs, 208–210, 210–211
  USB, 226–237
    implementing, 234–237
    packets, 229–231
    physical, 231–234
interference, 78
internal power-on reset
  ATtiny15 processor, 119
  PIC processors, 112
Interrupt Acknowledge (IACK), 164
interrupt service routine (ISR), 17
interrupts, 17–19
  DSP56805-based computers, 178–179
  hardware, 17–18
  software, 18
  transfers, 230
inverting input, 254
I/O, 26
  interrupts, 17–18
  memory-mapped, 11
  microcontroller, 9–11
  Motorola 68000, 164–165
  ported, 11
IR (infrared), 220–226
  devices, 225–226
  IrDA, 220–224
  IrDA interfaces, 224–225
isochronous transfer, 229
ISP (In-System Programming), 24
ISR (interrupt service routing), 17

J

JTAG (Joint Test Action Group), 175, 183

K

keepout layers, 84
Kirchhoff's Current Law, 41
Kirchhoff's Voltage Law, 42

L

LAN (Local Area Network), 238
languages
  assembly, 6–8
  selecting, 7
layers, software, 5
laying out PCBs, 87–90
LED (Light Emitting Diode), 53
  adding, 121–122
  potentiometer, 195
light sensors, 263–265
linear regulators, 70, 71
little-endian schemes, 16–17
LM78xx voltage regulators, 73
load/store architecture, 21
Local Area Network (LAN), 238
logic analyzers, 95–96
logic families, 59
logical address, 149
loop areas, minimizing, 79
looping, 22
low-pass filter, 47

M

MAC (Multiply and Accumulate) unit, 173
machine code, 6
magnetic field sensors, 270–271
management
  memory, 149–155
  voltage, 281–282
maps, memory, 11, 138–144
mask-programmable ROMs, 24
Master In Slave Out (MISO), 188
Master Out Slave In (MOSI), 188
MAX1615 voltage regulator, 74
MAX603 voltage regulators, 73
MAX604 voltage regulators, 73
MCLR, 112
mechanical layers, 84
memory, 22–26
  addresses, 11
  banked, 149, 151
  cycles, 132–134
  DMA, 26–32
  DSP56805-based computers, 179–183
  EEROM, 25
  EPROM, 25
  Flash, 25
  management, 149–155
  maps, 11, 138–144
  microcontroller, 9–11
  Motorola 68000, 165–166
  nonvolatile, 4
  RAM, 23–24
  ROM, 24
  sharing, 10
  SPI, 197–199, 200–201
  stacks, 15
  storage, 11
Memory Management Unit (MMU), 150
memory-mapped I/O, 11
MFP (Multifunction Peripheral), 161
Microchip software development environment (MPLAB), 109
microcontrollers, 9–11
  AVR (see entries under AVR)
  DSP56805, 175
  PIC (see entries under PIC)
microprocessors, 9–11
  buses, 11
  clocks, 56–59
MIMD (Multiple-Instruction Multiple Data) computers, 30–32
minimizing current loop areas, 79
MISO (Master In Slave Out), 188
MMU (Memory Management Unit), 150
mnemonics, 6–8
modem (MOdulator-DEModulator), 213
modules, oscillator, 57
monitoring
  light sensors, 263–265
  magnetic field sensors, 270–271
  motor speed, 281
  pressure sensors, 267–270
  temperature sensors, 261–263
MOSI (Master Out Slave In), 188
motor control, 274–281
Motorola 68000, 156–157
  address decoder, 162–164
  architecture, 157–161
  design, 161–169
  I/O, 164–165
  memory, 165–166
  reset circuit, 162
  wait states, 167–169
MPLAB (Microchip software development environment), 109
Multifunction Peripheral (MFP), 161

multimaster bus, 202
multimeter, 95–96
Multiple-Instruction Multiple-Data (MIMD) computers, 30–32
Multiply and Accumulate (MAC) unit, 173

N

negative current flow, 37
networks
  CAN, 241–245
  Ethernet, 245–251
  LAN, 238
  RS-485, 238–241
noise, 77–82
  RS-422, 218–220
noninverting input, 254
nonvolatile data memory, adding with SPI, 197–199
nonvolatile memory, 4
Nyquist frequency, 257

O

OE (output enable), 144
  read enable signals, 138
Ohm's Law, 40
One-Time Programmable (OTP) ROMs, 24
opcodes, 6, 7
operands, 13
operating systems, 4, 18
operational amplifiers, 253
organization of memory, 22
oscillators, 56–59
  DSP56805-based computers, 176
  PIC processors, 112
  SPI-based clock/calendar, 190–194
oscilloscope, 95
OTP (One-Time Programmable) ROMs, 24
output enable (OE), 144
  read enable signals, 138
output of PWM, 274
overlay layers, 84

P

packets, 229–231
pads, 85
PALs, 144–146
parallel ADCs, 256
parallel computers, 28–32
parallel/distributed, 28–32
parameter memory, adding with SPI, 200–201
parity of UARTs, 210
Pascals, 268
passive components, 42
  inductors, 49–51
  (see also capacitors; resistors)
PCB (printed circuit-board), 79, 83–87, 87–90
  building, 91–94
  testing, 96–99
peak-to-peak amplitude, 39
periodic signal intervals, 38
Peripheral Interface Controller (PIC)
  architecture, 107–109
  PIC12C508, 109–113
  PIC16C73, 113–114
peripherals
  ADC (see ADC)
  CAN, 241–245
  DAC, 271–273
  Ethernet, 245–251
  I2C, 201–205
    displays, 206
    RTC, 205
  IrDA, 220–224
  MFP, 161
  SPI, 187–201
    clock/calendar, 190–194
    memory, 197–199, 200–201
    potentiometer, 194–197
  UART (see UART)
  USB (see USB)
phantom signals, 79
physical address, 149
PIC (Peripheral Interface Controller)
  architecture, 107–109
  PIC12C508, 109–113
  PIC16C73, 113–114
piezoelectric, 55
plug packs, 68
popping values, 15
populating components, 88
POR (power-on reset), 112, 119
ported I/O, 11
ports, 63
  IR, 220–226
    devices, 225–226
    IrDA, 220–224
    IrDA interfaces, 224–225
  power, 64
  RS-232C, 211–218
    handshaking, 214–215
    implementing, 215–217
    power supply, 217–218

RS-422, 218–220
serial, 208–211
SPI, 125
UARTs, 208–210
error detection, 210–211
USB, 226–237
implementing, 234–237
interfaces, 231–234
packets, 229–231
positive current flow, 37
potentiometer, 194–197
power, 39
AVR processors, 116–118
batteries, 69–70
bricks, 68
DC, 68–69
ports, 64
reducing consumption, 58
regulators, 70–76
RS-232C interfaces, 217–218
sizing current loops, 79–81
testing, 100
power-on reset (POR), 112, 119
pressure sensors, 267–270
printed circuit-board (PCB), 79, 83–87, 87–90
building, 91–94
testing, 96–99
processors, 6–8
adding, 100
addressing modes, 15
ALU, 13–14
ATtiny15, 118–125
adding status LEDs, 121–122
switching analog signals, 122–125
AVR, 116–118, 127–128
ATMEL AT90S8535, 128
downloading code, 125–127
big endian/little endian, 16–17
DMA, 26–32
DSPs, 21, 170–172
DSP56800, 173–175
DSP56805, 175–183
interrupts (DSP56805), 178
JTAG, 183
memory (DSP56805), 179–183
oscillator (DSP56805), 176
RESET (DSP56805), 178
functions, 13
interrupts, 17–19
I/O, 26
memory, 22–26
EEROM, 25
EPROM, 25
Flash, 25
RAM, 23–24
ROM, 24
microcontroller, 9–11
microprocessors, 9–11
buses, 11
clocks, 56–59
Motorola 68000, 156–157
address decoder, 162–164
architecture, 157–161
design, 161–169
I/O, 164–165
memory, 165–166
reset circuit, 162
wait states, 167–169
PIC
architecture, 107–109
PIC12C508, 109–113
PIC16C73, 113–114
registers, 14
SHARC, 171
SOC, 9
soft core design, 144
stacks, 15
StarCore, 171
VLIW, 171
Program Controller, 173
program counters, 14
programs (see applications)
prototyping wirewrapping, 83
pulling values, 15
Pulse Width Modulation (PWM), 175, 273–274
pushing values, 15

Q

quantization, 256
quiescent current, 71

R

rail-to-rail operation, 254
RAM (Random Access Memory), 23–24
rates, sample, 257–259
RC circuits, 45–49
read enable signals, 138
output enable (OE), 144
Read-Only Memory (ROM), 24

Real-Time Clock (RTC), 191, 205
rectification, 53
reduction of power consumption, 58
registers, 13, 14
regulators, 70–76
Request To Send (RTS), 214
reset circuit
DSP56805-based computers, 178–179
Motorola 68000, 162
resistors, 39–42
RC circuits, 45–49
resources, xvi–xvii
return path, 79
RISC (Reduced Instruction Set Computer), 19–21
roll off, 253
ROM (Read-Only Memory), 24
routing
circuits, 91–94
PCBs, 83
RS-232C, 211–218
RS-422, 218–220
RS-485, implementing, 238–241
RTC (Real-Time Clock), 191, 205
RTS (Ready To Send), 217
RTS (Request To Send), 214

S

sample rates, 257–259
samples per second (SPS), 256
sampling, 256
schematics, 61–64
MAX1615 voltage regulator, 74
selection of languages, 7
self-powered devices, 231
semiconductors, 37
sensors
accelerometer, 266
light, 263–265
magnetic field, 270–271
motor speed, 281
pressure, 267–270
temperature, 261–263
Serial Peripheral Interface (SPI), 125, 187–201
clock/calendar, 190–194
memory, 197–199, 200–201
potentiometer, 194–197
serial ports
CAN, 241–245
IR, 220–226
devices, 225–226
IrDA, 220–224
IrDA interfaces, 224–225
RS-232C, 211–218
handshaking, 214–215
implementing, 215–217
power supply, 217–218
RS-422, 218–220
RS-485, 238–241
UARTs, 208–210
error detection, 210–211
USB, 226–237
implementing, 234–237
interfaces, 231–234
packets, 229–231
series connections, 41
(see also capacitors; resistors)
shadow registers, 14
SHARC processors, 171
sharing memory, 10
signals
amplifiers, 252–255
analog, 38–39, 122–125
bus, 134–138
converting, 271–273
digital, 59–61
avoiding noise, 77–82
logic analyzers, 95–96
phantom, 79
sample rates, 257–259
silkscreen layers, 84
SIMD (Single-Instruction Multiple Data) computers, 29
sinking current, 59
slaves, 189
small-signal resistance, 55
SOC (System-On-Chip), 9
soft core processor designs, 144
software
interrupts, 18
layers, 5
solder mask, 84
soldering, 96–99
sources
of power, 68–69
sourcing current, 59
speed sensors, 281
SPI (Serial Peripheral Interface), 125, 187–201
clock/calendar, 190–194
memory, 197–199, 200–201
potentiometer, 194–197
SPS (samples per second), 256

SRAM (static RAM), 22
SSI (Synchronous Serial Interface), 175
stacks, 15
standard block transfer, 27
star ground point, 261
StarCore processors, 171
START condition, 202
static operation, 58
static RAM (SRAM), 22
status LEDs, 121–122
status register, 14
storage, memory, 11
stubs, 90
stuffing components, 88
switching
  analog signals, 122–125
  regulators, 70, 71
synchronous processors, 133
Synchronous Serial Interface (SSI), 175
system architecture
  buses, 11
  DMA, 26–32
  Harvard, 11
  I/O, 26
  memory, 22–26
    EEROM, 25
    EPROM, 25
    Flash, 25
    RAM, 23–24
    ROM, 24
  processors
    addressing modes, 15
    ALU, 13–14
    big endian/little endian, 16–17
    functions, 13
    registers, 14
    stacks, 15
  Von Neumann machine, 10–11
  (see also architecture)
system software, 6
System-On-Chip (SOC) processors, 9

T

tantalum capacitors, 45
temperature sensors, 261–263
testing
  circuits, 96–99
  PCBs, 96–99
  power, 100
  processors, 100
  (see also debugging)
timing analysis, 146–148
timing diagrams, 128–132
title block, 61
tools
  debugging, 95–96
  (see also sensors)
tracks, 84
transfers, 26–32
transformers, 51–52
Transistor-Transistor Logic (TTL), 60
transitions, timing diagrams, 130
translation of addresses, 154
transmissions
  IR, 220–226
    devices, 225–226
    IrDA, 220–224
    IrDA interfaces, 224–225
  RS-422, 218–220
  UARTs, 210
  USB, 226–237
    implementing, 234–237
    interfaces, 231–234
    packets, 229–231
traps, 17
trimmer, 194–197
tristate output, 59
troubleshooting
  debugging, 95–96
  electromagnetic effects, 77–82
  ESD, 81
TTL (Transistor-Transistor Logic), 60
twisted pair, 218
types
  of ADCs, 256
  of capacitors, 44–45
  of diodes, 55
  of DSPs, 171
  of PCBs, 84
  of regulators, 70
  of schematics, 61

U

UART (Universal Asynchronous Receiver Transmitter), 208–210
  during construction, 210
  error detection, 210–211
unipolar signal, 38
Universal Asynchronous Receiver Transmitter (UART), 208–210
  error detection, 210–211

Universal Serial Bus (USB), 226–237
  implementing, 234–237
  interfaces, 231–234
  packets, 229–231
upstream connections (USB), 232
USB (Universal Serial Bus), 226–237
  implementing, 234–237
  interfaces, 231–234
  packets, 229–231

V

vacuum tubes, 253
value of stacks, 15
valves, 253
variable-speed oscillator (PIC), 112
varying voltage, 40
vectored interrupts, 18
VLIW (Very Long Instruction Word) processors, 171
volatile memory, 23
voltage, 36–38
  amplifiers, 252–255
  analog
    converting, 255–259
    signals, 38–39
  capacitors, 43–45, 45–49
  constant, 40
  crystals, 55–59
  DC, 68–69
  divider, 41
  Ethernet, 246–251
  inductors, 49–51
  managing, 281–282
  potentiometer, 195
  power, 39
  PWM, 273–274
  regulators, 70–76
  resistors, 45–49
  RS-232C, 211–218
  RS-422/RS-485, 219
  transformers, 51–52
  varying, 40
Von Neumann machine architecture, 10–11

W

wait states, 133
  Motorola 68000, 167–169
watchdog, 119
Watts, 39
watts
  (see also power)
wire
  inductors, 49–51
  transformers, 51–52
wirewrapping, 83
working register (PIC), 108

X

XON/XOFF, 215

Z

zero-overhead looping, 22

About the Author

John Catsoulis is an electronics engineer, programmer, and physicist who specializes in advanced computer architectures. He is responsible for the design of over 25 embedded computer systems, and since 1996 has been Managing Director of Embedded Pty Ltd., a company that designs computers for industry, government, military, and scientific agencies.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *Designing Embedded Hardware* is a porcelain crab. These tiny invertebrates are common in tide pools along the coast of the Pacific Ocean. They are an orangy brown color, and are only 5 millimeters long. Porcelain crabs have six pairs of legs, with one tiny pair tucked in across the base of the tail. Although they can swim, sharply pointed spines on the ends of their walking legs make it easier for them to cling to the hard surfaces of submerged rocks. Hair on their legs collects mud from the ocean floor and helps camouflage the crab from predators. Additional protection is provided by mussel beds, sponges, and algae. Once concealed in these preferred habitats, the porcelain crab sweeps its feathery arms through the water, capturing plankton and other tiny plants and animals. When threatened by a predator, these crabs can detach a leg or claw to distract an attacker. The tricky crab scurries away, and its lost appendage eventually grows back.

Philip Dangler was the production editor for *Designing Embedded Hardware*. Norma Emory was the copyeditor. Argosy provided production services and wrote the index. Sheryl Avruch and Jane Ellin provided quality control.

Emma Colby designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from *Cuvier's Animals*. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted from XML to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Philip Dangler.

About the Author

John Catsoulis is an electronics engineer, programmer, and physicist who specializes in advanced computer architectures. He is responsible for the design of over 25 embedded computer systems, and, since 1996, has been Managing Director of Embedded Pty Ltd., a company that designs computers for telecoms, government, military, and scientific agencies.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *Designing Embedded Hardware* is a porcelain crab. These tiny invertebrates are common in tide pools along the coasts of the Pacific Ocean. They are an orangy brown color, and are only 5 millimeters long. Porcelain crabs have six pairs of legs, with one tiny pair tucked in across the base of the tail. Since they can't swim, sharply pointed spines on the ends of their walking legs make it easier for them to cling to the hard surfaces of submerged rocks. Flat, shallow legs collect food from the ocean floor and helps camouflage the crab from predators. Additional protection is provided by mussel beds, sponges, and algae. Once concealed in these protective habitats, the porcelain crab sweeps its feathery arms through the water, capturing plankton and other tiny plants and animals. When threatened by a predator, these crabs can detach a leg or claw to distract an attacker. The tricky crab scurries away, and its lost appendage eventually grows back.

Philip Dangler was the production editor for *Designing Embedded Hardware*. [illegible] Emory was the copyeditor. Argosy provided production services and wrote the index. Sheryl Avruch and Jane Ellin provided quality control.

Emma Colby designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from Cuvier's Animals. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted from SGML to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Philip Dangler.

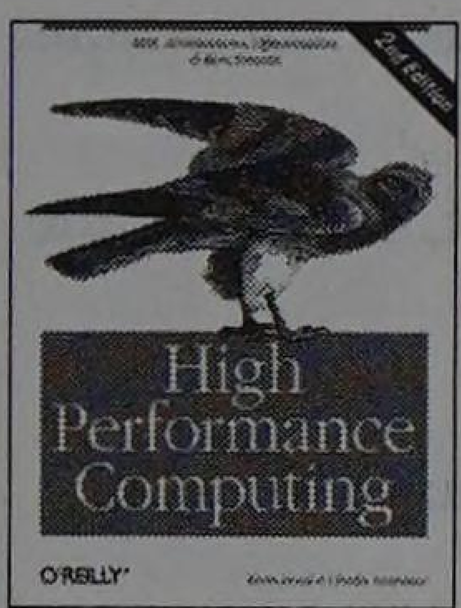
Other Titles Available from O'Reilly

C and C++ Programming

C++: The Core Language

By Gregory Satir & Doug Brown
1st Edition October 1995
228 pages, ISBN 1-56592-116-X

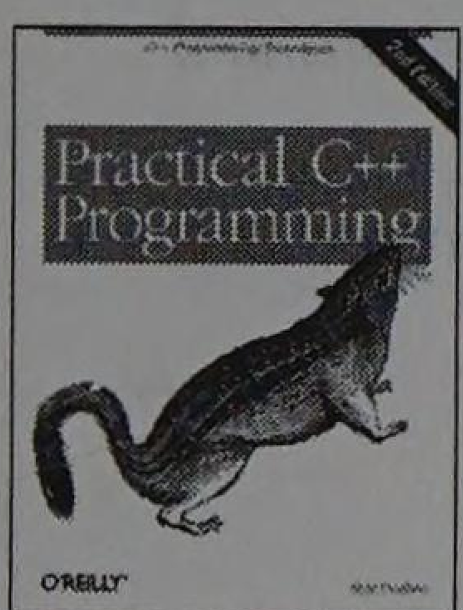
A first book for C programmers transitioning to C++, an object-oriented enhancement of the C programming language. Designed to get readers up to speed quickly, this book thoroughly explains the important concepts and features and gives brief overviews of the rest of the language. Covers features common to all C++ compilers, including those on Unix, Windows NT, Windows, DOS, and Macs.



High Performance Computing, 2nd Edition

By Kevin Dowd & Charles Severance
2nd Edition July 1998
466 pages, ISBN 1-56592-312-X

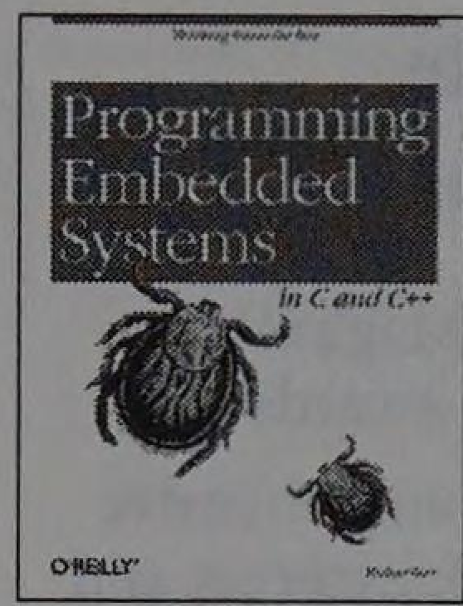
This new edition of *High Performance Computing* gives a thorough overview of the latest workstation and PC architectures and the trends that will influence the next generation. It pays special attention to memory design, tuning code for the best performance, multiprocessors, and benchmarking.



Practical C++ Programming, 2nd Edition

By Steve Oualline
2nd Edition December 2003 (est.)
600 pages (est.), ISBN 0-596-00419-2

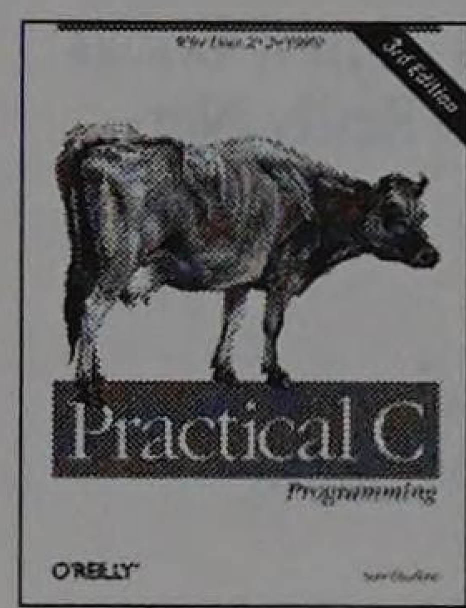
Practical C++ Programming is a complete introduction to the C++ language for programmers who are learning C++ or transitioning from C to C++. In a new edition reflecting the latest changes to the C++ standard, this book takes a practical, real-world approach, placing a strong emphasis on coding style and the pro-gramming process. Readers learn not only the syntax of C++, but also how to write clear, maintainable code and to design and debug programs in a systematic way. For a hands-on approach to learning C++, this is the book to recommend.



Programming Embedded Systems in C and C++

By Michael Barr
1st Edition January 1999
194 pages, ISBN 1-56592-354-5

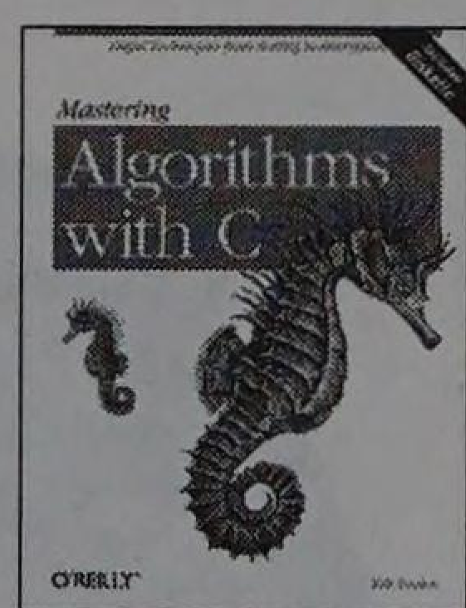
This book introduces embedded systems to C and C++ programmers. Topics include testing memory devices, writing and erasing Flash memory, verifying nonvolatile memory contents, controlling on-chip peripherals, device driver design and implementation, optimizing embedded code for size and speed, and making the most of C++ without a performance penalty.



Practical C Programming, 3rd Edition

By Steve Oualline
3rd Edition August 1997
454 pages, ISBN 1-56592-306-5

Practical C Programming teaches you not only the mechanics of programming, but also how to create programs that are easy to read, maintain, and debug. This third edition introduces popular Integrated Development Environments on Windows systems, as well as Unix programming utilities, and features a large statistics-generating program to pull together the concepts and features in the language.



Mastering Algorithms with C

By Kyle Loudon
1st Edition August 1999
560 pages, Includes CD-ROM
ISBN 1-56592-453-3

This book offers robust solutions for everyday programming tasks, providing all the necessary information to understand and use common programming techniques. It includes implementations and real-world examples of each data structure in the text and full source code on the accompanying disk. Intended for anyone with a basic understanding of the C language.

How to stay in touch with O'Reilly

1. Visit our award-winning web site

http://www.oreilly.com/

★ "Top 100 Sites on the Web"—PC Magazine
★ CIO Magazine's Web Business 50 Awards

Our web site contains a library of comprehensive product information (including book excerpts and tables of contents), downloadable software, background articles, interviews with technology leaders, links to relevant sites, book cover art, and more. File us in your bookmarks or favorites!

2. Join our email mailing lists

Sign up to get email announcements of new books and conferences, special offers, and O'Reilly Network technology newsletters at:

http://elists.oreilly.com

It's easy to customize your free elists subscription so you'll get exactly the O'Reilly news you want.

3. Get examples from our books

To find example files for a book, go to:

http://www.oreilly.com/catalog

select the book, and follow the "Examples" link.

4. Work with us

Check out our web site for current employment opportunities:

http://jobs.oreilly.com/

5. Register your book

Register your book at:

http://register.oreilly.com

6. Contact us

O'Reilly & Associates, Inc.
1005 Gravenstein Hwy North
Sebastopol, CA 95472 USA
TEL: 707-827-7000 or 800-998-9938
(6am to 5pm PST)
FAX: 707-829-0104

order@oreilly.com
For answers to problems regarding your order or our products. To place a book order online visit:

http://www.oreilly.com/order_new/

catalog@oreilly.com
To request a copy of our latest catalog.

booktech@oreilly.com
For book content technical questions or corrections.

corporate@oreilly.com
For educational, library, government, and corporate sales.

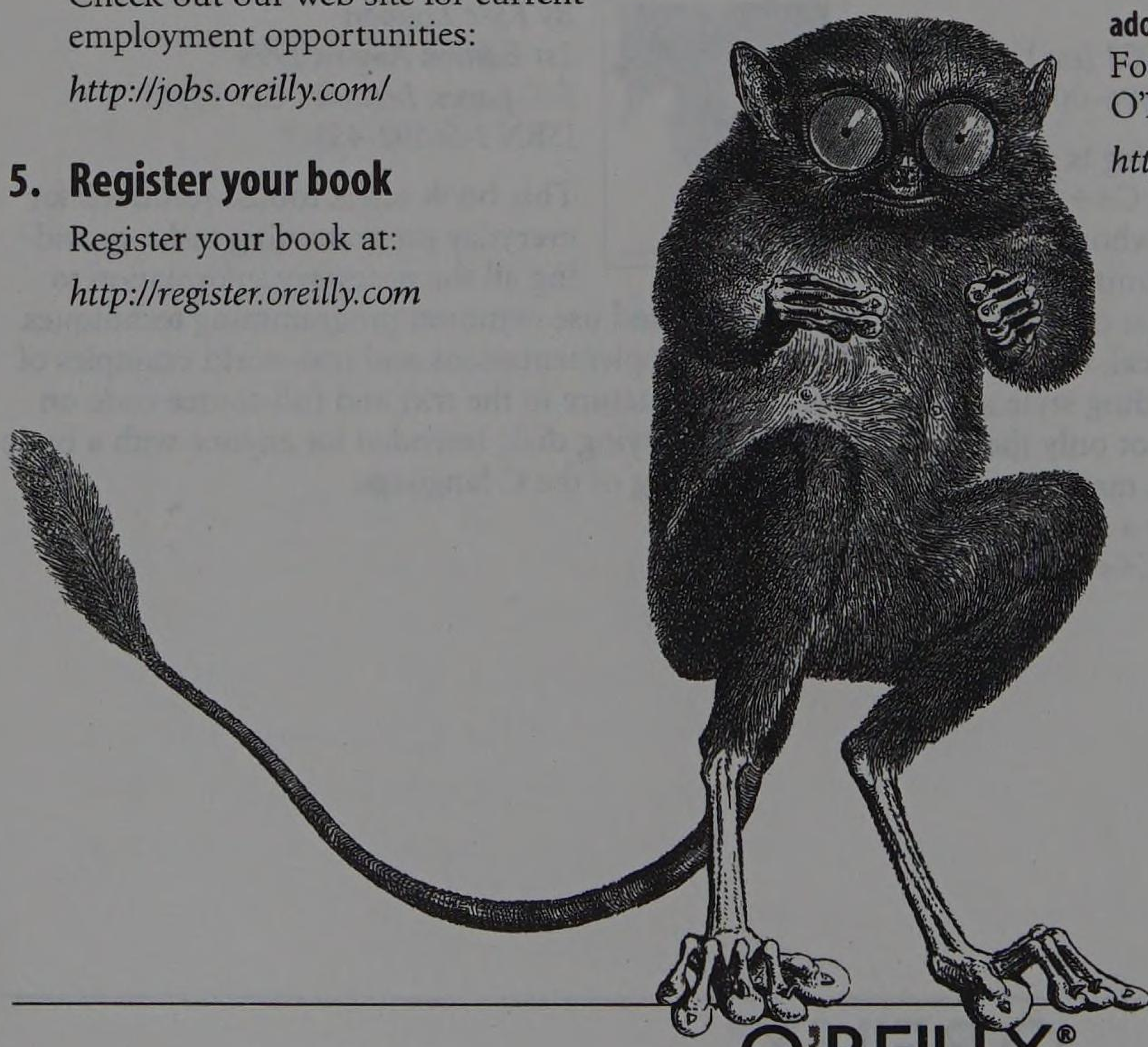
proposals@oreilly.com
To submit new book proposals to our editors and product managers.

international@oreilly.com
For information about our international distributors or translation queries. For a list of our distributors outside of North America check out:

http://international.oreilly.com/distributors.html

adoption@oreilly.com
For information about academic use of O'Reilly books, visit:

http://academic.oreilly.com



O'REILLY®

To order: *800-998-9938* • *order@oreilly.com* • *www.oreilly.com*
Online editions of most O'Reilly titles are available by subscription at *safari.oreilly.com*
Also available at most retail and online bookstores.

O'REILLY®

Designing Embedded Hardware

You use embedded computers every day, whether it's the system controlling your toaster, your alarm clock, or your car's automatic transmission. Experienced programmers know that knowledge of the underlying hardware is critical for crafting the best embedded software.

Designing Embedded Hardware is a book about designing small machines for embedded applications. There are many books on the market dedicated to writing code for particular microprocessors, or that stress the philosophy of embedded system design without providing any practical information. This book steers a middle path, telling you what you need to know to create your own products, and distilling much of the lore of embedded systems design into a single volume. It shows you how to build a complete embedded system, add peripherals, and connect your system to other devices.

Designing Embedded Hardware covers:

- The theory and practice of embedded systems
- Powering an embedded system
- Producing and debugging an embedded system
- Processors such as the PIC, Atmel AVR, and Motorola 68000-series
- Digital Signal Processing (DSP) architectures
- Protocols (SPI and I^2C) used to add peripherals
- RS-232C, RS-422, infrared communication, and USB
- Networks (RS-485, CAN, and Ethernet)

Software professionals who want to design their own hardware—not assemble a PC, but create entirely new devices and computerized gadgets—will find a wealth of information in this book to help them penetrate the mysteries of creating hardware.

US $39.95
CAN $61.95
ISBN 0-596-00362-5
90000
9 780596 003623
6 36920 00362 5

Visit O'Reilly on the Web at *www.oreilly.com*